University of Alberta

Parallel Fault Simulation on the C•RAM Architecture

by

Albert L.C. Kwong

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-34387-1

Canada

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Parallel Fault Simulation on the C•RAM Architecture** submitted by Albert L.C. Kwong in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Bruce F. Cockburn

Dr. Duncan G. Elliott

Dr. Martin Margala

Dr. Jonathan Schaeffer

Date: Oct 1st, 1998

Abstract

Fault simulation is used to determine if the faulty behaviour of a given circuit with a fault added is detectable when a list of test patterns is supplied. Simulating all potential faults in a circuit with a sufficient number of patterns to observe all faults is computationally intensive. Fortunately, a high degree of parallelism exists in the fault simulation application. For example, conventional parallel fault simulation can take advantage of the datapath of the processor and use the logical operations provided by the CPU to perform parallel gate evaluations. This algorithm can be further accelerated using a massively parallel computer.

C•RAM is a SIMD architecture that integrates processing elements with memory. A system with 256M bytes of C•RAM memory can contain up to 128k processing elements (PE), providing a very wide datapath. We wished to determine whether or not a parallel fault simulator implemented on C•RAM could make effective use of this datapath to accelerate fault simulation.

In this research, three different experimental C•RAM fault simulators were designed. The simulators implement pattern-parallel fault simulation (*simf_pps*), fault-parallel fault simulation (*simf_fps*), and a hybrid of the pattern-parallel and fault-parallel algorithms (*simf_hps*). Two versions of *simf_hps* were implemented: static-hybrid parallel fault simulation and dynamic-hybrid parallel fault simulation.

By using the ISCAS 85 benchmark circuits to evaluate the simulators, it was found that all of our simulators (except *simf_fps*) run faster than our benchmark conventional simulator, *simf*. *Simf_fps* is generally inefficient, except for circuits with mostly easy-to-detect faults. *Simf_pps* is not as efficient as the hybrid fault simulators

in most of the cases, except when the circuit has many hard-to-detect faults. When simulating with 4096 PEs, the hybrid fault simulators were found to improve upon our benchmark simulator by 10 to 30 times. The hybrid fault simulator is scalable in that we observe increasing speed-up for up to 16k PEs.

Acknowledgements

I would like to thank my supervisors, Dr. Bruce F. Cockburn and Dr. Duncan G. Elliott, for their help in the course of this research. They have provided valuable ideas and background knowledge. I would also like to thank my supervisory committee, Dr. Martin Margala and Dr. Jonathan Schaeffer, who have given me many useful comments. My thanks also goes to TRLabs and Micronet for their financial support.

Most importantly, I thank God who gave me this opportunity to study and the ability to finish it. Praise the Lord. Amen.

Contents

1 Introduction **1**

1.1 Problem Definition 1

1.2 Testing and Fault Simulation 2

1.3 SIMD Parallel Architectures 4

1.4 Thesis Outline 5

2 C•RAM and SIMD machines **7**

2.1 History 7

2.2 The C•RAM Architecture 10

2.2.1 The Processing Element (PE) 11

2.2.2 The Communication Network 12

2.2.3 Memory Structure 14

2.3 The C•RAM Programming Model 14

2.4 Advantages and Disadvantages of C•RAM 17

3 Fault Models **19**

3.1 Introduction 19

3.2 Stuck-At Faults 20

3.3 Gate-Delay Faults 22

3.4 Stuck-Open Faults 25

3.5 Fault Implication and Equivalence Among Fault Models 28

4 Fault Simulation **36**

4.1 Introduction 36

4.2 Sequential Fault Simulation 40

4.3 Fault-Parallel Fault Simulation 43
4.4 Pattern-Parallel Fault Simulation 44
4.5 Hybrid Parallel Fault Simulation 46
4.6 Evaluation of Fault Simulators 47
4.6.1 Faults, Undetected Faults, and Triggered Faults 47
4.6.2 Measuring Efficiency - PE Utilization 49
4.6.3 Another Measurement - Speed-up 49

5 simf - a Fast Conventional Pattern-Parallel Fault Simulator 51
5.1 Overview 51
5.2 Design Details 52
5.2.1 Circuit Data Structure 53
5.2.2 Fault Generation 55
5.2.3 Simulation Algorithm 58
5.3 Evaluation of simf 69

6 Pattern-Parallel Fault Simulation on C•RAM 72
6.1 Overview 72
6.2 Design Details 73
6.2.1 Data Structure 73
6.2.2 C•RAM Assembler 75
6.3 Evaluation 80
6.4 Discussion 82

7 Fault-Parallel Fault Simulation on C•RAM 84
7.1 Overview 84
7.2 Design Details 86
7.2.1 Simulation with C•RAM 86
7.2.2 Fault Simulation 90
7.3 Evaluation 98
7.3.1 Test Results 98
7.3.2 Optimization with Fault Grouping 100
7.4 Discussion 101

8 Hybrid Fault Simulation on C•RAM **103**

8.1 Overview 103

8.2 Design Details 105

8.2.1 Partitioning C•RAM Memory 105

8.2.2 Modified Fault-Family Fault Simulation 109

8.2.3 Hybrid-Parallel Fault Simulation 112

8.2.4 Fault Triggering and Fault Detection 113

8.2.5 Dynamic-Hybrid Fault Simulation 115

8.3 Evaluation 118

8.3.1 PE Utilization 121

8.3.2 Simulating Using a Larger C•RAM 125

9 Conclusion **128**

9.1 Summary of Results 128

9.2 Further Research 130

9.2.1 Fault-Simulating Large Circuits 131

9.2.2 Sequential Circuits 131

9.2.3 Other Fault Models 132

Bibliography **133**

Appendices **136**

A Source Code **136**

A.1 Basic Modules 136

A.1.1 structure.h 136

A.1.2 prototype.h 137

A.1.3 simf.C 137

A.1.4 read_iscas.C 138

A.1.5 gen_fault.C 141

A.1.6 fault_family.C 145

A.1.7 heap.h 146

A.1.8 heap.C 147

A.1.9 cram_mapper.h 148
A.1.10 cram_mapper.C 148
A.2 simf Benchmark Fault Simulator 150
A.3 Pattern-Parallel Fault Simulator 156
A.4 Fault-Parallel Fault Simulator 161
A.5 Hybrid Fault Simulator 166

List of Tables

3.1 Triggering Conditions for Stuck-At Faults affecting NOT Gates (Inverters) 21
3.2 Triggering Conditions of Stuck-At Faults for 2-input NAND Gate . . 24
3.3 Triggering Conditions for Gate-Delay Faults Affecting a 2-input NAND Gate 27
3.4 Triggering Conditions for Stuck-Open Faults Affecting 2-input AND Gate 28
3.5 Triggering Conditions for Faults Affecting the NOT Gate 30
3.6 Triggering Conditions for Faults Affecting the BUFF Gate 30
3.7 Triggering Condition for Faults Affecting the NAND Gate 31
3.8 Triggering Conditions for Faults Affecting the AND Gate 32
3.9 Triggering Conditions for Faults Affecting the NOR Gate 33
3.10 Triggering Conditions for Faults Affecting the OR Gate 34

5.1 Root-to-Fault Ratio for the ISCAS85 Circuits 68
5.2 Evaluation of Simulation Speed (in seconds) 69
5.3 Fault Collapsing Results for the ISCAS85 Circuits 70

6.1 Comparing *simf* and *simf_pps* in Terms of Speed-Up 80
6.2 Comparison of PE Utilization for *simf* and *simf_pps* 81

7.1 Speed Difference in Fault-Free Simulation 86
7.2 Memory Requirement With or Without Dynamic Memory Allocation 88
7.3 Interpretation of Flags in the Event-List 93
7.4 Result Comparison of PPS with FPS 97
7.5 Analysis of the Fault-Parallel Fault Simulation Results 99

7.6 PE Utilization : Average Number of FFs Simulated Per Pass 99
7.7 Effect of Depth-First-Search from Output Optimization 100
7.8 Fault Triggering Detection Time in FPS 101

8.1 Speeding-up HPS by Using Depth-First-Search Grouping 110
8.2 Speeding-up HPS by Using C•RAM Mapper 112
8.3 Final Results 118
8.4 PE Utilizations of the Hybrid Fault Simulators 123
8.5 Fault Simulation Results with 4K PEs 125
8.6 Speed-up of Fault Simulations Going from 1K PEs to 4K PEs 125

9.1 Optimization Implemented in Each Fault Simulator 129
9.2 Fault Simulation Results With 4K PEs Running at 143MHz 130

List of Figures

2.1 SISD - Single Instruction Stream, Single Data Stream 7
2.2 SIMD - Single Instruction Stream, Multiple Data Stream 8
2.3 MIMD - Multiple Instruction Stream, Multiple Data Stream 9
2.4 The C•RAM Architecture 10
2.5 C•RAM PE Structure 11
2.6 C•RAM Programming Models 15
2.7 Memory Bandwidth Throughout the System 17

3.1 Node Types 20
3.2 Inverter (NOT gate) with Possible Stuck-At Faults 21
3.3 Fault Collapsing for the Stuck-At Fault Model 22
3.4 Gate-Delay Faults Associated with 2-input NAND Gates 23
3.5 Fault Collapsing for the Gate-Delay Fault Model 25
3.6 A 2-input NAND Gate with Stuck-Open Fault 26
3.7 Common Structure of CMOS AND Gates and OR Gates 27
3.8 Fault Implication and Fault Collapsing Relationships 35

4.1 The Scan Chain/**BIST** Architectures 36
4.2 Hardware Configuration of RTPGs 38
4.3 Fault Coverage of Different RTPGs 39
4.4 Flow Chart of the Basic Fault Simulation Algorithm 41
4.5 2-Dimensional Fault Simulation Space 42
4.6 2-D Fault Simulation Space of Fault-Parallel Fault Simulation 44
4.7 2-D Fault Simulation Space of Pattern-Parallel Fault Simulation . . . 45
4.8 2-D Fault Simulation Space of Hybrid Parallel Fault Simulation . . . 47
4.9 Classes of Faults with Respect to Fault Detection 48

4.10 Curves for Undetected Faults, Triggered Faults and Detected Faults . 48

5.1 Top Level Architectural Design of *simf* 51
5.2 Computer Representation of Circuit Nodes in PPSFP 53
5.3 Top Level View of the Circuit Data Structure 54
5.4 A Fault-List Example 56
5.5 Gate Evaluation in *simf* 59
5.6 Fault Propagation in Circuits 60
5.7 Using a Heap for Sorting Simulation Events 62
5.8 Heap - Random Input, Sorted Output 63
5.9 Detecting Circuit-Node Transitions 65
5.10 Fault Insertion Using Transition Masks 66

6.1 Data Structure Used for Storing Circuit Node States 73
6.2 Using the First PE to Backup Data 74
6.3 Copying Test Patterns into C•RAM 76

7.1 Dataflow Diagram of *simf_fps* 85
7.2 Illustration of Circuit Node Life-Cycles 87
7.3 Gate Evaluation with Multiple Fault-Insertion 91
7.4 Problem with Fan-out Nodes 91
7.5 Fault Insertion With the Token-passing Method 95

8.1 Static-Hybrid Fault Simulation versus Dynamic-Hybrid Fault Simulation 104
8.2 Flow Chart of a Fault-Simulation Round 106
8.3 Logical Partition of C•RAM 107
8.4 Partitioning With PEid 108
8.5 Data Structures Supporting Fault Grouping 109
8.6 Simulating With Fault-Groups in a C•RAM Partition 111
8.7 Fault Triggering/Detection Mechanism 113
8.8 Choosing the Best Fault-to-Pattern Ratio 117
8.9 Effect of Using Different Number of Partitions (Normalized to f=1) . 120
8.10 Determining a Useful Set of PEs 122
8.11 PE Utilization Curve 124

8.12 Speed-Up Graphs for c2670nr, c3540nr, c6288nr, and c7552nr 127

Glossary

equivalent faults Two faults are equivalent if they produce identical behaviour from the primary inputs to the primary outputs.

erroneous node A node is said to be erroneous if its Boolean state in fault simulation is complementary from that in a good simulation.

fanout node A node that contains at least three pieces of wire: the stem wire that comes out of a gate output, and the two or more fanout wires that leads to other gate inputs.

fanout wire A wire in a fanout node that is connected to a gate input.

fault or logical fault Abstract representation of the effects on signals of one or more physical defects in the circuit under test.

fault collapsing The process of reducing the number of faults by finding faults that are equivalent and then only retaining one fault from each fault equivalence class.

fault coverage The ratio between the number of detected faults and the total number of considered faults in a circuit under test.

fault detection The situation when the effects of a fault propagate to at least one primary output and thus produce an externally observable error.

fault dropping The process of removing detected faults from the fault list.

fault effect The consequence of a fault which causes a wire to contain complementary values in a modeled faulty circuit and the fault-free circuit.

fault implication Fault A implies fault B if the detection of fault A by a test implies that fault B will also be detected by the same test.

fault list At the start of a fault simulation, this list contains all considered faults. The list is updated to contain only undetected faults before each round of fault simulation.

fault model Used in a general sense to mean the set of all the different faults used to represent all possible faulty behaviours in a circuit under test.

fault propagation The process of determining the spread of fault effects downstream from a fault going towards the primary outputs of the circuit under test.

fault simulation The process of computing the future behaviour of a modeled faulty circuit under test when presented with a given test.

fault triggering A process which determines whether or not the conditions exist for creating the primary effect of a fault in the circuit under test.

fault type Faults are sometimes grouped into fault types according to the kind of family behaviour involved. In this thesis, we consider the following fault types: stuck-at-0 (SA0), stuck-at-1 (SA1), slow-to-rise (SR), slow-to-fall (SF), N-transistor stuck-open (NO), P-transistor stuck-open (PO).

good simulation The process of determining the node values in a fault-free circuit under test when presented with a given test.

ISCAS85 circuits A set of benchmark combinational circuits in a specific format that is widely used in the research community. They were released to the research community at the 1985 International Symposium on Circuits and Systems.

node a model of the low resistance connection of one or more physical conductors in a circuit that should carry, in a good circuit, the same electrical signal.

node (non-fanout) A node that contains only one piece of wire. The wire connects one gate output with one gate input.

PE utilization The ratio between the number of PEs doing useful work and the total number of PEs in the C•RAM. A PE does useful work if the corresponding calculation would have to be done in a conventional sequential fault simulator.

physical defect A manufacturing flaw in integrated circuits that can change the behaviour of a circuit.

primary effect The first fault effect at a gate output that is caused in a circuit under test by the presence of a physical defect.

primary input A circuit node that is driven by a signal source external to the circuit under test.

primary output A circuit node that does not drive other gates in the circuit under test and whose state is observable by an external tester or observer.

simulation pass The process in which a subset of the currently undetected faults are fault simulated together with a subset of the test patterns.

simulation round A sequence of simulation passes that fault simulates all the undetected faults for a given set of test patterns.

stem wire A wire in a fanout node that is connected to a gate output.

test A sequence of test patterns that is intended to be used to detect most if not all considered faults in a circuit under test.

test pattern A Boolean input vector that is used as part of a test. The length of a test pattern equals the number of primary inputs.

test pattern generation The process of producing a sequence of test patterns to be used as a test. Test pattern generation can be implemented either using custom hardware or as a program running on a programmable computer or tester.

simf The benchmark conventional pattern-parallel fault simulator used in the thesis.

simf_fps A fault-parallel fault simulator implemented on C•RAM.

simf_pps A pattern-parallel fault simulator implemented on C•RAM.

wire A model of a physical conductor in a circuit. Wires are assumed to be indivisible or primitive elements of a circuit.

Chapter 1

Introduction

1.1 Problem Definition

When manufacturing Very Large Scale Integration (VLSI) chips, random defects occur despite the best precautions. To insure that shipped chips are defect-free, it is necessary to apply a sequence of test patterns[1] to the inputs of the circuit under test (CUT) and then compare the output signals from the CUT with those expected of a good chip. A defect is detected if the sequence of outputs generated by the CUT is different from that of a good chip. Different sequences of test patterns can detect different faults. It is desirable to select a minimum length sequence of test patterns to detect most, if not all, of the expected faults since the test time is proportional to the number of test patterns required for testing, and testing is a major cost of production.

Fault simulations are run to evaluate the fault detecting ability of a proposed sequence of test patterns. This involves taking the sequence of test patterns, possibly generated by a pseudo-Random Test Pattern Generator (RTPG), and inputting the patterns into a simulated CUT. In a *fault simulation*, the circuit is simulated with a fault inserted, a fault that is selected from one of many possible associated fault models. By "inserting a fault" it is meant that the circuit is assumed to be affected by the fault. In a *fault-free simulation*, on the other hand, the CUT is assumed to have no faults at all, i.e. the CUT is assumed to be good. The output signal sequence from a fault simulation is compared with that of the fault-free one to determine if the presence of the inserted fault can be observed as a signal at at least one CUT output

[1] A sequence of n bit binary numbers where n equals the number of primary inputs.

that differs from the corresponding signal produced by a good CUT.

The main goal of the research described in this thesis is to evaluate alternative parallel implementations of well-known parallel fault simulation algorithms on the massively parallel Computational RAM (C•RAM) architecture [7].

1.2 Testing and Fault Simulation

Physical defects can cause a chip to behave in numerous different erroneous ways. For example, a piece of wire that is accidentally shorted to ground will appear to be stuck at the logic value 0, assuming positive logic encoding. Physical defects are typically too complex and too numerous to consider directly, therefore simplified Boolean fault models are introduced [14]. The effects of the physical defects are generalized and grouped into various simplified fault models. For example the *stuck-at fault* represents all the defects that could cause a piece of wire to appear to be stuck at a logical 1 or 0.

In the past, digital VLSI manufacturers only tested for stuck-at faults, the most simple fault model. Stuck-at faults affecting combinational circuits are detectable using single test patterns. However, as the quality requirements for VLSI chips have continued to increase, it has also become necessary to test for more complex sequential faults[2], such as gate delay faults and stuck-open faults. A *gate delay fault* assumes that the physical defect will cause a signal to be driven from low-to-high or high-to-low overly slowly. A *stuck-open fault* assumes that a transistor channel or wire connected to it is disconnected permanently. These kinds of faults require two consecutive test patterns to be applied in order for the fault to be detected. Therefore, the pattern space for detecting these faults is roughly the square of that of the stuck-at faults and more effort is needed to search for a sequence of test patterns that yields 100% fault coverage.

As mentioned earlier, it is necessary to generate a sequence of test patterns that detects most of the expected possible faults in a given circuit. There are two basic ways of doing so. One way is to compute them *deterministically* from the structure of the circuit. A memory unit is required for storing each precomputed test sequence

[2]Faults that cause a combinational circuit to exhibit memory.

before applying them to a CUT. As the size of the circuit increases, the size of the memory required also increases. Another way of generating the test sequence is to use pseudo-random test pattern generators (RTPG). Some well known RTPGs are the Linear Feedback Shift Register (LFSR) and the Linear Hybrid Cellular Automata (LHCA) [4]. Different RTPGs use different algorithms to generate a pseudo-random signal sequence. Fault simulation is needed to evaluate alternative RTPG algorithms.

It is necessary to distinguish between single-fault simulation and multiple fault simulation. The *single-fault assumption* implies that any circuit under test will contain a maximum of one fault; on the other hand, the *multiple fault assumption* implies that a circuit can contain any number of faults. In reality, the multiple fault assumption is likely to be able to represent real defects more accurately than the single-fault assumption; however, simulating for multiple faults of unlimited multiplicity requires the simulation of a very large number of circuits, approximately proportional to the factorial of the number of wires in the CUT. In [17] it is found that a sequence of test patterns that detects all single stuck-at faults has a 99.9% chance of detecting all the multiple stuck-at faults. Similar research on other fault models has generally confirmed the effectiveness of the single-fault assumption. Therefore, it is reasonable to omit multiple-fault simulations. In this research, only single-fault fault simulations are considered.

Fault simulation can take a very long time to execute. Harel and Krishnamurthy have shown that there exists a class of circuits that requires $O(n^2)$ simulation time, where n is the number of gates [13]. Several efficient fault simulation acceleration techniques have been proposed, such as the parallel, and concurrent techniques. *Concurrent fault simulation* is currently (as of 1998) the most popular technique because it is efficient and it supports not only the simulation of logic units but also of more complex functional blocks [30]. *Parallel fault simulation*, although it doesn't support complex blocks as well as the concurrent technique, is as efficient as concurrent simulation. Since our goal was to use C•RAM to accelerate the simulation process, parallel fault simulation, due to its parallel computational structure, should fit nicely onto C•RAM, which is a SIMD architecture. Thus, parallel fault simulation was selected for implementation on the C•RAM architecture. The rest of this chapter provides additional background on testing, fault simulation, and relevant parallel computer

architectures.

1.3 SIMD Parallel Architectures

The SIMD (*Single-Instruction stream, Multiple-Data stream*) computing model was defined by Flynn in 1966 [10]. In this computing model, a control unit issues a single stream of instructions to an array of identical processors, called the processing elements (PEs). Each processing element executes this instruction stream in parallel, on their own data in a local memory; hence we have multiple data streams.

Historically, SIMD machines have often been characterized by their high computational power and high price tag. These machines include the Distributed Array Processor (DAP) designed at Active Memory Technology Inc. [26], Thinking Machines Corporation's Connection Machine (CM-2) [29], and MasPar Computer Corporation's MP-1 [2]. They have been used mainly by government and research institutions [22] for scientific and military research.

While all of the above machines are substantial computer systems, there are also SIMD computers that are based on integrated SIMD processors and on-chip main memory. This category of SIMD computers are becoming more popular. The main driving force for this change in technology was the advance in VLSI processing technology. These new designs can integrate on one chip the processing elements together with the memory, thus taking advantage of the high data transmission rate (bandwidth) inside the chip. Recent custom chip designs include the processing chip of BLITZEN designed at the Microelectronics Center of North Carolina [15], Texas Instruments' Serial Video Processor (SVP) [5], NEC Corporation's Integrated Memory Array Processor (IMAP) [33] and Parallel Image-Processing RAM (PIP RAM) [1], the Processing-In-Memory (PIM) chip designed at the Supercomputing Research Center [12], and Sony Corporation's Linear Array Architecture DSP [19]. These processor-memory integrated circuits are sometimes called Processors-In-Memory (PIM) or Intelligent RAM (IRAM) [27].

One design is the *Computational RAM* (C•RAM) developed at the University of Toronto [8]. It is characterized by a large number of processing elements (PEs) and memory elements that are integrated into one chip. Basically, the C•RAM

architecture is a modified Random Access Memory (RAM). RAM chips, including both Dynamic-RAM (DRAM) and Static-RAM (SRAM), are organized into rows and columns of memory cells. Each memory cell can store one bit of data. In the C•RAM architecture, several columns of memory cells are grouped together with a single PE integrated at their sense amplifiers[3] (where the number of columns in a group depends on the implementation). The columns of memory thus become the PE's local memory. C•RAM was designed so that the area overhead of the processing elements is minimized, thus achieving a high memory-per-PE ratio. One unique feature of the C•RAM is that, besides being a SIMD machine, the memory elements of a C•RAM chip can also be used as in a normal memory chip, thus possibly acting as the main memory of a computer system.

1.4 Thesis Outline

There are two basic ways to implement parallel fault simulation. One way is to simulate a number of faulty circuits in parallel with one common test pattern. This is called *fault-parallel fault simulation*. The other way is to simulate a number of test pattern in parallel for one common faulty circuit. This is called *pattern-parallel fault simulation*. There is a very popular fault simulation technique based on the pattern-parallel approach called the Parallel Pattern Single Fault Propagation algorithm (PPSFP) [31] [32]. A program (*simf*) based on this scheme was implemented for this thesis, and its performance was compared with another efficient public domain fault simulator, named *sim3*, from the University of Victoria. The new PPSFP program was then used as a reference to evaluate our C•RAM fault simulators. Three versions of fault simulators that run on C•RAM were developed. One uses the pattern-parallel technique (*simf_pps*), one uses the fault-parallel technique (*simf_fps*), and the last one is a hybrid of the two (*simf_hps*). The ISCAS85 benchmark circuits were used as the CUTs [3].

The goal of this thesis is to evaluate three alternative ways to parallelize fault simulation running on the C•RAM architecture. This thesis is divided into the following chapters: Chapters 2 to 4 are background sections that introduce the SIMD architec-

[3]A sense amplifier amplifies the weak recovered cell signal to a full-strength logical 0 or logical 1.

ture, fault modeling and fault simulation, respectively. The design of the programs are discussed in Chapters 5 to 8. The ISCAS file format, common data structures for each program, and the PPSFP simulator running on a sequential machine are discussed in Chapter 5. Chapter 6 presents the design of the pattern-parallel fault simulator running on the C•RAM architecture. Chapter 7 describes the fault-parallel version, and Chapter 8 the hybrid version. Finally, conclusions are made in Chapter 9.

Chapter 2

C•RAM and SIMD machines

2.1 History

By referring to how the instruction streams and data streams are handled, computer architectures can be divided into four categories, namely, Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD) [10]. SISD machines (shown in Figure 2.1) are conventional machines with a single processor executing a single instruction stream. When a number of SISD machines are put together to execute a single instruction stream on their own data, we have a SIMD architecture. A MIMD machine is one with multiple processing units executing their own instruction streams on their own data, and occasionally communicating among themselves. Finally, a MISD machine is one that executes multiple instruction streams on a single data stream. Whether this architecture exists is debatable in the computer architecture community.

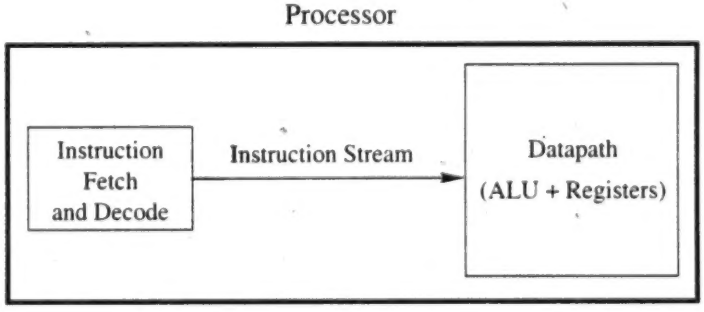


Figure 2.1: SISD - Single Instruction Stream, Single Data Stream

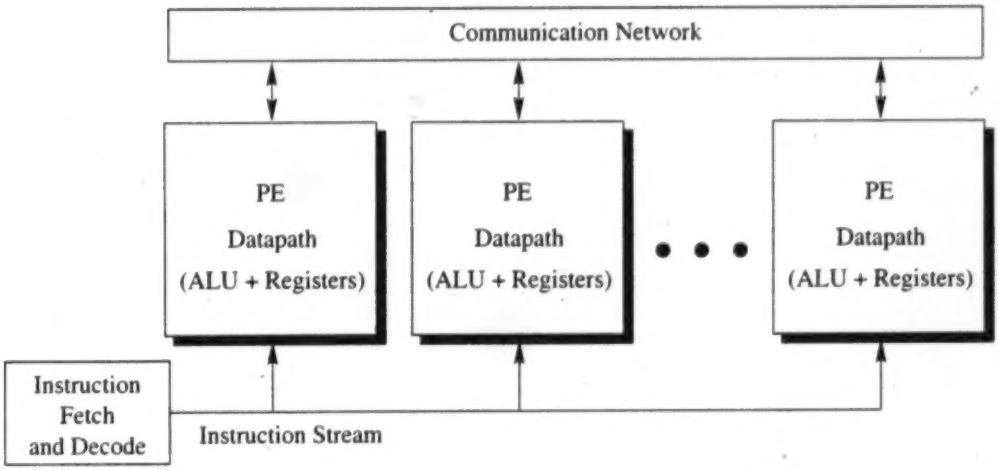


Figure 2.2: SIMD - Single Instruction Stream, Multiple Data Stream

In the early days of computing, the majority of computers were SISD machines. To achieve higher computing power than a SISD machine could deliver, a logical way is to group many SISD machines together. In the 1950s, since each transistor was so precious, it was necessary to achieve the highest possible parallelism with the lowest hardware overhead. Under these constraints, the SIMD architecture was invented. A SIMD machine (shown in Figure 2.2) contains an array of identical processors, called the *processing elements* (PE). These PEs execute the same instruction stream in parallel on their local data stream. SIMD machines have usually been extremely expensive because of the high cost of hardware, design, and programming (which requires a good compiler). As a result, the main market for these machines has, in the past, been largely limited to government research agencies.

Although a massively parallel SIMD architecture offers potentially superior performance over a conventional SISD architecture, it can generally only do so for a certain type of application. For an application to take advantage of the parallelism of a SIMD machine, the problem must be decomposed into an array of similar sub-problems. Each of these sub-problems must be mapped onto a PE of the SIMD machine so that multiple sub-problems can be executed in parallel. Not all applications can be conveniently decomposed in this way; thus, in the early days, these machines were usually used for scientific applications, that can be expressed in a

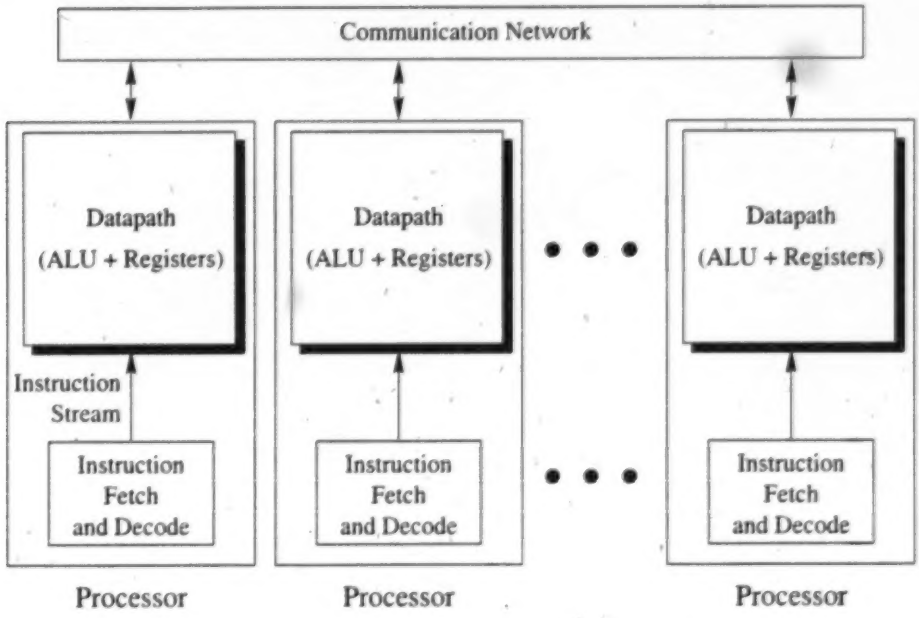


Figure 2.3: MIMD - Multiple Instruction Stream, Multiple Data Stream

matrix or vector format.

In a MIMD architecture (shown in Figure 2.3), each processor has its own instruction stream (instruction fetch and instruction decode) and data stream. Processors have to exchange messages (or use share memory) among each other for such purposes as sharing intermediate results and other data. This kind of communication requires extra hardware, which was costly when transistor density was precious. It is also desirable to use the MIMD architecture because modern operating system design favors a multi-process environment, where more than one process can run in parallel. The parallel processes can be mapped onto different processors of the MIMD machine. Since it is often usually easier to map parallel problems into parallel processes, this type of architecture has become more popular [25].

SIMD architecture designers took advantage of advances in VLSI technology. In earlier days, SIMD machines were built using processors and commodity memory modules. The main drawback of this design practice is that memory access is relatively slow because both the processor and the memory chips have to use an external data bus to communicate. To solve this problem, several manufactures have designed

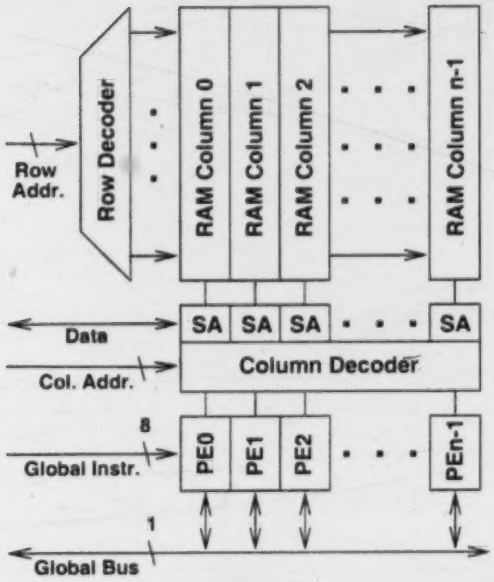


Figure 2.4: The C•RAM Architecture

single chip SIMD machines. One example of such a design is Texas Instruments' Serial Video Processor (SVP) [5]. The SVP chip was designed as a video signal processor for processing digital TV signals. It contains a serial input port and a serial output port, 1024 PEs, and 256 bits of memory per PE. Since the processing elements and the memory both reside in the same chip, the memory access time is greatly reduced. Other chips include NEC Corporation's Integrated Memory Array Processor (IMAP) [33] and Parallel Image-Processing RAM (PIP RAM) [1]. These chips were designed to target digital image processing. Another chip that integrates processor and memory is the Processor-In-Memory (PIM) chip designed at the Supercomputer Research Center [12].

2.2 The C•RAM Architecture

Having described trends in the computing industry with respect to SIMD architectures, it is time to discuss the design that will be used in this research project, namely the Computational RAM (C•RAM) architecture [7]. As shown in Figure 2.4, the C•RAM architecture is essentially a DRAM chip with Processing Elements attached to the sense amplifiers at the bottom of the memory columns to provide parallel

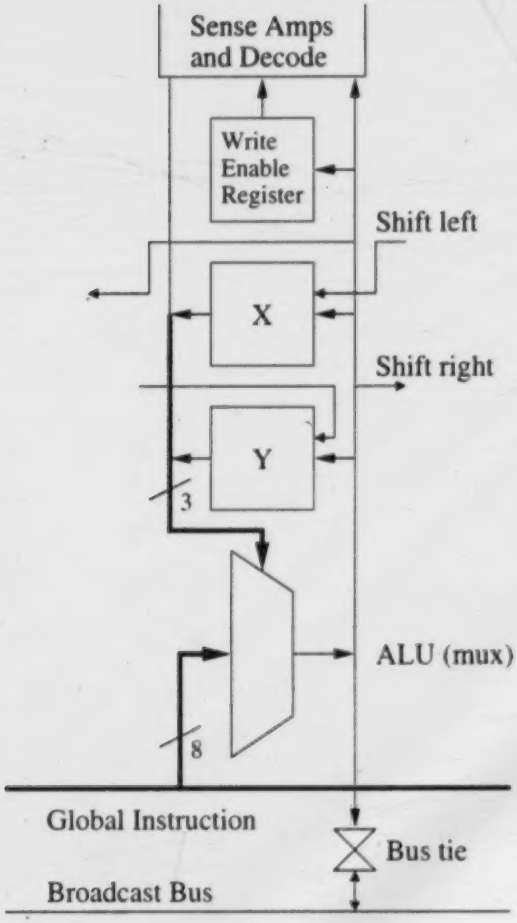


Figure 2.5: C•RAM PE Structure

processing. Traditionally, SIMD machines differ in the design of the PEs, the communication network, and the memory structure. Hence the C•RAM architecture will be described in terms of these three parameters.

2.2.1 The Processing Element (PE)

There are a variety of SIMD PE designs. One basic difference is the word size of each PE. NEC Corporation's Integrated Memory Array Processor (IMAP), for example, uses a 8-bit processing element [33]. The chip contains 64 such processing elements with 2 Mb of memory. Each processing element in this design consists of an 8-bit arithmetic logic unit (ALU), an 8-bit shifter, 12 registers and a 4-bit exchange unit,

which adds up to about 7000 transistors. This is an example of a complex SIMD PE design.

The C•RAM PE architecture, on the other hand, is an extremely simplified SIMD design. As shown in Figure 2.5, C•RAM PE consist of an 8-1 multiplexer (MUX) and three single bit registers: *X*, *Y*, and *write-enable*. The *write-enable* register is used as a mask so that a subset of PEs can write back to their local memory while the local memory of the other PEs is not affected. In normal operation, every PE's *write-enable* register is set to logical 1. The 8-1 MUX, using the X register, Y register, and selected memory[1] as input selects[2], serves as an ALU. With this configuration, any 3-variable truth table can be treated as an opcode[3] and thus an instruction set with 256 opcodes is implemented. This PE design is very simple, thus requiring little chip area. For a given die size, we can implement more PEs or more memory compared to complex PE designs.

2.2.2 The Communication Network

It is often necessary to move data among PEs through a communication network. Traditionally the communication network has been a big concern of SIMD manufacturers. For example, the MasPar MP-1 architecture provides routing nodes to route data between PEs in data packets (Multistage Crossbar Interconnect) in addition to the basic nearest neighbor communication mechanism (X-Net Mesh Interconnect) [2]. Another classical design, Thinking Machine's CM-2, implements a Hypercube Interconnection Network with a maximum distance of 12 hops between any two of the 4096 PEs [29]. These complicated communication mechanisms are aimed at reducing the communication time between PEs for arbitrary PE-generated destinations.

In other SIMD designs, the design of the communication network has been greatly simplified. In the SVP chip, each PE is capable of communicating directly with its nearest neighbors and the nearest neighbors' nearest neighbors [5]. In other words, each PE is capable of communicating directly with its four nearest neighbors in a one-dimensional array. This kind of design is especially useful when the chip is used

[1] For each C•RAM operation, a row of memory is selected

[2] The input selects of a multiplexer determine which input signal appears at the multiplexer output.

[3] An opcode, short for operation code, is a machine instruction.

for video signal processing, where digital filters may be applied to the incoming video signal for image enhancement, such as gamma correction [5].

Other designs, such as the Terasys, implement more complicated communication mechanisms [12]. The designer of this chip provides five ways for inter-processor communication, including linear interconnection, global-OR, partitioned-OR, parallel prefix networks (PPN), and data movement through the host processor.

The design of the communication network in C•RAM is very similar to that of PIM, except that C•RAM does not have either the PPN or the partitioned-OR implemented. The designer of C•RAM, Duncan Elliott, believes that a segmented wired-OR bus can emulate the PIM PPN without loss of parallelism. The global-OR mechanism ties the output of each PE's ALU to a bus in a wired-OR fashion. In other words, the output of the ALUs can be logically ORed together with a single instruction. This network is very powerful for algorithms that need to compare values between all the PEs, such as the problem of finding the minimum value among all PEs, in which case an algorithm of order *O(log2(n))* could be used [20]. An alternative implementation of this network is the global-AND network[4]. In our research, the global-AND implementation was used.

Another communication mechanism, as mentioned earlier, is linear interconnect. Linear interconnect connects each PE with its two nearest neighbors, in a one-dimensional array. This design is the same as the SVP's except that the SVP connects the four nearest neighbors instead of two. The linear interconnection network is useful for linear array algorithms that repeatedly require data from their left and right neighbors, such as digital filtering. The leftmost and the rightmost PEs have only one neighbor, and these PEs can be configured to communicate with PEs in another C•RAM chip. A bank of C•RAM chips can thus be connected in a two-dimensional fashion with very little extra hardware.

The last type of communication mechanism involves moving data to and from C•RAM memory using the host processor. Moving data via this method is relatively slow as it doesn't take advantage of the high data bandwidth internal to the C•RAM chip. However the data transfer rate is identical to reading one or more words from ordinary memory and then writing them back into another memory location.

[4]The bus equals 1 if and only if all PEs output a 1.

2.2.3 Memory Structure

Memory structure is another important part of SIMD architectures. Two issues related to memory design are discussed here. The first is *shared memory*. In most SIMD architectures, each PE has its own data area called local memory. Shared memory is a data area that more than one PE can access directly. Some architectures, such as the MasPar MP-1, implement shared memory as one way to communicate data between PEs [2]. Implementing shared memory involves resolving conflicts when more than one PE tries to access the same memory area, thus increasing the complexity of the PE.

Another issue concerning memory is whether or not PEs should access their local memory with *indirect addressing*. By indirect addressing, we mean that during a local memory access cycle, each PE can access its local memory with a different offset. For example, during a memory access cycle, PE no.1 may be accessing data located at the 5th bit of its local memory, while PE no.2 accesses the 20th, and PE no.3 accesses yet another memory location. Indirect addressing is very useful in terms of programming flexibility, however, it would also imply a more complicated memory structure where each bank of local memory has its own addressing hardware.

C•RAM aims at providing the highest number of memory and processing elements within a chip, while providing enough features for implementing parallel algorithms with reasonable efficiency. With this aim in mind, the memory structure was kept simple, and neither shared memory nor indirect addressing were implemented. A regular DRAM (Dynamic Random Access Memory) array is used as the local memory block for the PEs. The local memory of each PE can span several columns of DRAM memory. In one particular design, for example, a PE is inserted for every four columns of DRAM cells, adjacent to the column sense amplifiers [7]. The local memory for each PE is thus composed of four columns from the DRAM cell array.

2.3 The C•RAM Programming Model

There are basically two programming models for the C•RAM machine (See Figure 2.6. The first, the *bit-parallel* model, is to combine multiple PEs to form a processor with a bigger word size. For example, eight PEs can be grouped together as an 8-

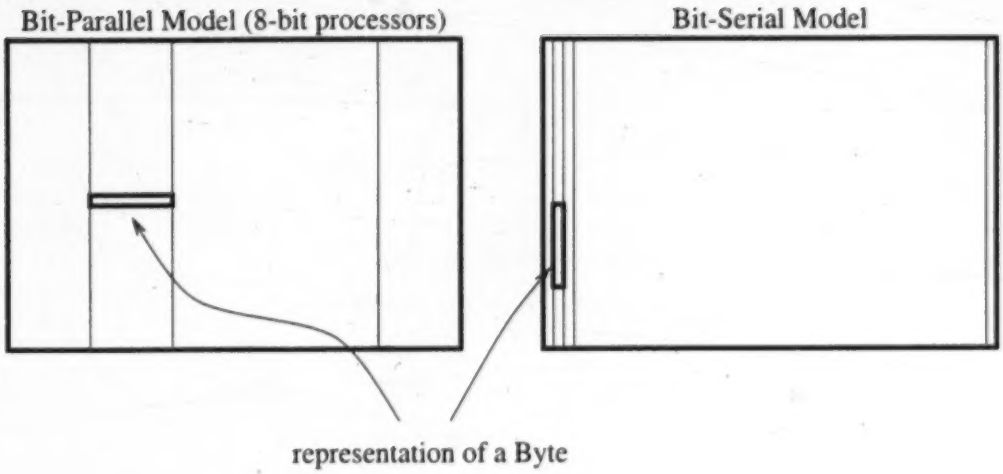


Figure 2.6: C•RAM Programming Models

bit processing unit. In this bit-parallel model, an 8-bit word is stored horizontally occupying one memory cell in the local memory of each PE (of the grouped processing unit). The vector quantized image compression algorithm was programmed this way, with a modified C•RAM architecture [20]. Another programming model, the *bit-serial* model, is to view each PE individually as a single-bit processor. In this model, an 8-bit word is stored vertically in one PE's local memory, occupying 8 memory cells per PE. Figure 2.6 demonstrates the difference between these two models. In this research, we are going to use the bit-serial programming model.

The C•RAM programming environment is a C++ library that contains a C•RAM emulator [7]. This emulator not only emulates the functionality of the hardware but also provides timing information. Three C•RAM classes are defined in this library for accessing the C•RAM. They are the `cboolean` (C•RAM Boolean), the `cint` (C•RAM integer), and the `cuint` (C•RAM unsigned integer).

The `cboolean` class defines a single bit variable that allocates one bit in every PE's local memory. A high-level flow control statement, named `cif`, is defined to take advantage of the *write-enable* register for conditional execution using a `cboolean` variable. The `cint` and `cuint` classes are basically the same except that one is signed and the other is unsigned. These classes are used to define multiple bit variables which occupy multiple bits in each PE's local memory. A library of high-level arith-

metic operations for `cint` and `cuint`, such as addition and subtraction, is provided to ease C•RAM programming. Another high-level construct is array subscripting, which allows an individual PE to be addressed. For example, A[20]=5 (where A is a `cint`/`cuint`) will load the A variable on the 19th PE (index starts at 0) with a value of 5.

These high-level library functions were built using the C•RAM assembly language. C•RAM assembly language also allows programmers to emulate the C•RAM directly. Use of these low-level operations can result in optimizations that permit faster execution. A variable declaration and a sample C•RAM assembly statement is shown as follows:

```
cint b(size);
b.operate (offset, opcode, output_mode [, flag]);
```

where the variable *b* is declared as a `cint` with *size* being the number of bits for the integer. For example, if *size* equals to 8, then 8 bits of memory per PE are allocated to this variable. The statement `b.operate`, is the assembly operation that tells the C•RAM to perform a specific instruction. The operation requires three parameters plus an optional one as shown above. The *offset* parameter specifies a bit in *b*, which corresponds to a row of memory cells. Thus if *b*'s size is 16 bits, then the range of *offset* can be anywhere from 0 to 15. This bit is fetched from each PE's local memory and is fed into the ALU as one of the three input selects. The other two input selects are the contents of the X and Y registers. The second parameter, *opcode*, is an 8-bit word corresponding to one of the 256 C•RAM instructions. This 8-bit word is used as the data input vector of the 8-1 MUX ALU. The last parameter, *output_mode*, instructs the C•RAM where the ALU output should be stored. Possible *output_modes* include the X register, the Y register, memory (specified by *offset*), the *write-enable* register, the left neighbor, the right neighbor, and the wired-AND network. In cases where the wired-network is chosen, the optional parameter *flag*, which is a variable of Boolean type, must be specified so that the wired-AND result can be stored. When supplying a *flag*, its value should typically be set to true, otherwise the returned result will always be false (because the value of the *flag* is also ANDed with the bus's state).

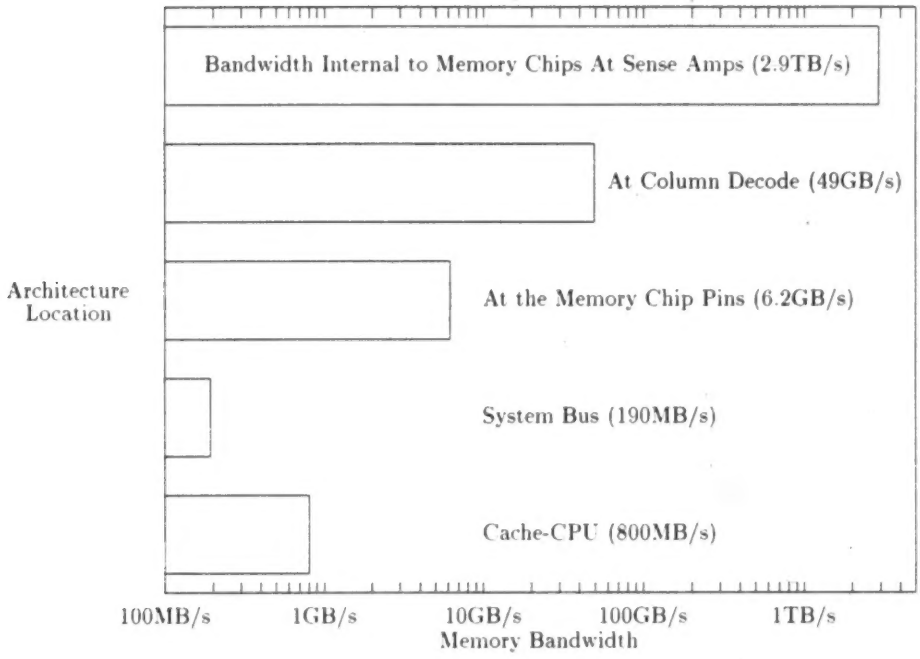


Figure 2.7: Memory Bandwidth Throughout the System

2.4 Advantages and Disadvantages of C•RAM

Many SIMD machines in computing history have provided tremendous processing power. However, they are usually very costly. The primary advantage of the C•RAM architecture is the availability of cheap massive parallelism. The PE of C•RAM is so simple that it can be implemented using a single layer of metal in circuit level [7]. This property is also possible in the DRAM. Since a large portion of the C•RAM circuit is DRAM, C•RAM could be manufactured in a DRAM fabrication facility using a DRAM process. These facilities usually produce high volumes of chips with relatively lower cost because the cost of memory is cheaper and C•RAM is mostly memory.

Despite its low cost, the C•RAM also provides high computing power. Since both the processing elements and memory reside in the same chip, the time for driving data signals along wires (the bit-lines) is minimized. When accessing memory, usually a row

of memory is selected from the 2D memory array, and then a small group of columns is selected, and the data is then sent to the output. The bandwidth available from the rest of the columns not selected is wasted. The C•RAM architecture avoids this waste by having its PEs located at the tips of the column sense amplifiers. The data items in one selected row are sent to their respective PEs, and parallel operations can make use of these data. The data bandwidth internal to the C•RAM is compared to the data bandwidth at various location in a microprocessor system in Figure 2.7 [7].

Another advantage of using C•RAM arises from its programming kit. The kit is built using the C++ programming language, which is very popular in today's computing industry, thus the learning curve of programming C•RAM is not as steep as in most of some other SIMD machines. Since the C•RAM architecture is pretty much self-contained, it can be used in both embedded applications and also as the main memory of workstations or personal computers. Combining these factors, C•RAM provides a convenient platform for scientific research as well as practical solutions to commercial problems.

C•RAM is not perfect. One main disadvantage is that the communication network supported is limited. Besides broadcasting data using the global data bus, each PE can only communicate with its nearest neighbors. If each PE has to pass a byte (8 bits) of data to its 4th neighbor to the right side, then a total of 32 instructions are needed to accomplish this task. This cost affects our choices of algorithms when designing C•RAM applications. The single bit data output is another bottleneck.

The fact that C•RAM supports only global memory addressing is another disadvantage. With this limitation, we cannot implement look-up tables efficiently. This again limits our choices of algorithms.

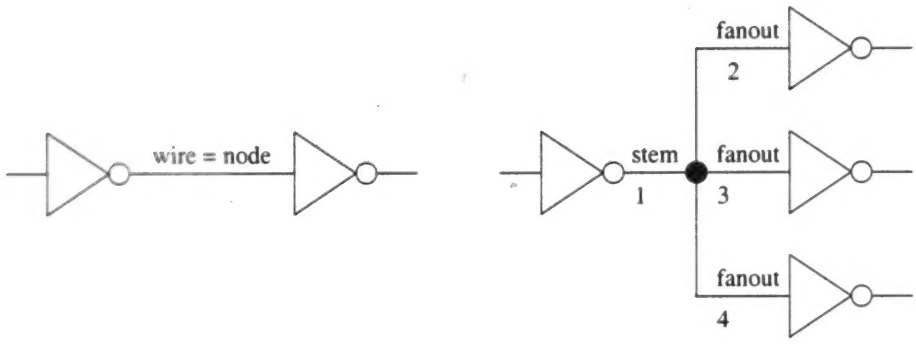
Chapter 3

Fault Models

3.1 Introduction

During the integrated circuit manufacturing process, physical defects are introduced as a result of contaminants and manufacturing flaws. These defects include short circuits, open circuits, incorrect component size, defective transistors, etc. Each of these defects affects the circuit functions in various ways. For example, if a 2-input OR gate contains a defective transistor, the net result could be that the gate output responds only very slowly to changes in input signals, or that it always gives a logical 0 as output. The number of ways that defects could cause a circuit to behave incorrectly is enormous, and it is thus very difficult to design tests to target all the possible defects in a circuit. An alternative way is to group the possible defects into classes of similar behaviours called *fault models*. For example, it is observed that many kinds of defects cause a circuit node to be logically stuck at 0, so a logical stuck-at 0 fault on this circuit node is used to represent these defects. The *stuck-at fault model* models the effects of all the defects that cause any circuit node to appear as if it is stuck at a logical 0 or logical 1. If a test sequence is capable of detecting all the stuck-at faults, then it is also capable of detecting all the physical defects that cause a stuck-at fault.

The fault models of interest in this research are the *stuck-at fault model*, the *gate-delay fault model*, and the *CMOS transistor stuck-open fault model*. Each of these fault models defines a number of faults equal to approximately the number of wires in a circuit. In reality, multiple faults can co-exist in a single circuit; however, if a circuit has n wires, the number of possible combinations of multiple faults is approximately 2^{3*n}, an extremely large number. It is nearly impossible to simulate



(a) Non-fanout node
1 wire in 1 node

(b) Fanout node
4 wires in 1 node

Figure 3.1: Node Types

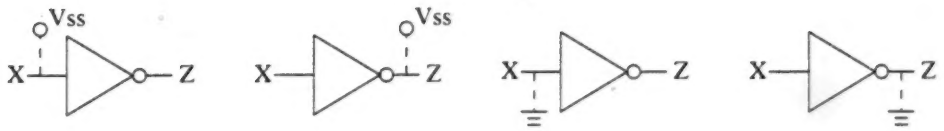
all combinations of multiple faults in a real circuit. In [17] it was shown that a test that detects all the single stuck-at faults could detect more than 99.9% of multiple stuck-at faults. Although similar research for the other 2 models has not been conducted, we believe that the single fault assumption for the other 2 fault models should be equally effective. Therefore in this research, we assume that any fault circuit contains exactly one fault of one type.

The rest of this chapter introduces the three fault models used in this research, including the behavior of the faults being modeled, the conditions for triggering these faults, and ways to reduce the number of faults that need to be considered explicitly.

3.2 Stuck-At Faults

The most widely used fault model is the stuck-at model. The stuck-at fault model assumes that each piece of wire in the circuit has the possibility of having a stuck-at 1 fault or a stuck-at 0 fault. A wire is not necessarily the same as a circuit node. For example, a *fanout node* contains at least three pieces of wire: the stem wire that comes out of a gate output, and the two or more fanout wires that leads to other gate inputs. Since each circuit wire can be stuck at either logic 1 or 0, the number of possible stuck-at faults in a circuit is equal to two times the number of wires in a circuit. The difference between wires and nodes is illustrated in Figure 3.1.

In order to detect a stuck-at 1 (0) fault for a piece of wire, the test sequence has to



X Stuck-at 1 SA1(X) | Z Stuck-at 1 SA1(Z) | X Stuck-at 0 SA0(X) | Z Stuck-at 0 SA0(Z)

Figure 3.2: Inverter (NOT gate) with Possible Stuck-At Faults

NOT (input X, output Z)

Fault	Input X	Good Z	Faulty Z
SA0(X)	1	0	1
SA1(X)	0	1	0
SA0(Z)	0	1	0
SA1(Z)	1	0	1

Table 3.1: Triggering Conditions for Stuck-At Faults affecting NOT Gates (Inverters)

first force the wire to a logical 0 (1). This is called *triggering* the fault. The erroneous output must then be *propagated* to at least one observing point. If this fault affects any of the observing points, then this fault is detected. So, triggering a fault is a precondition of detecting it.

Fault collapsing is the process of reducing the number of faults by finding faults that are *equivalent* and then only considering one fault in each fault equivalence class. Two faults are said to be equivalent if they can't be distinguished during testing. For example, the NOT gate in Figure 3.2 has two stuck-at faults at its input X, named SA0(X) and SA1(X), and two stuck-at faults at its output Z, named SA0(Z) and SA1(Z). From Table 3.1, we see that in order to trigger SA0(Z), the test sequence has to force input X to logical 0, and output Z is expected to produce a logical 1. This condition is the same as that for triggering SA1(X). Both of these faults would force Z to an erroneous logical 0. When the fault is detected, we simply can't tell whether the fault is a SA1(X) or a SA0(Z), therefore, we say that they are equivalent. Since the two faults are equivalent, only one of them needs to be considered for test pattern generation and fault simulation. In other words, we collapse the two faults into one. In fact, for the stuck-at fault model, we can say that any two faults that

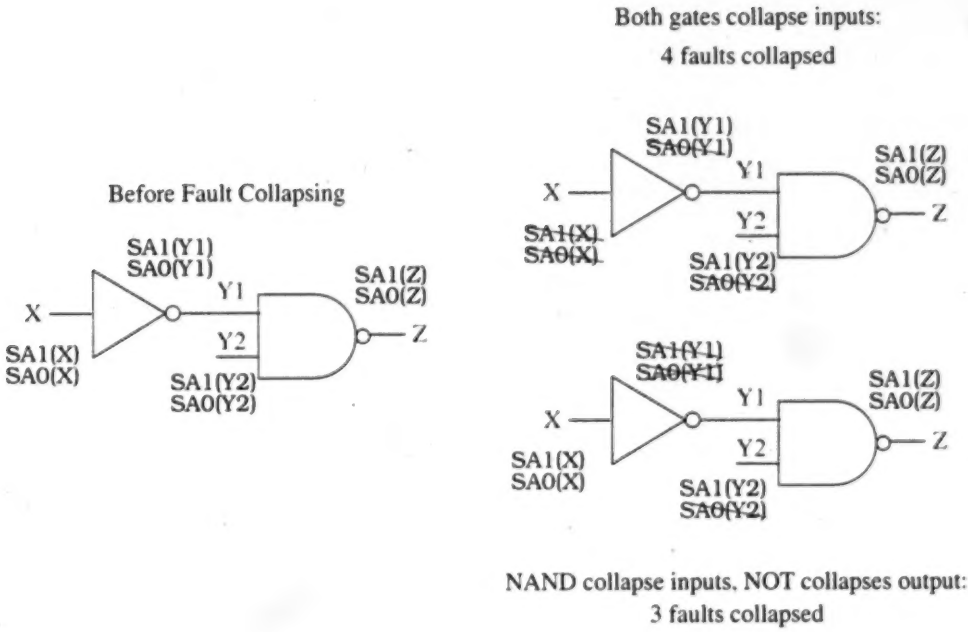


Figure 3.3: Fault Collapsing for the Stuck-At Fault Model

have the same triggering conditions are equivalent, and thus can be collapsed. The triggering condition for the faults of the other basic logic gates (BUFF, AND, OR, NAND, NOR) are listed in Tables 3.6 to 3.10 from [34].

Note that for inverters and buffers, the choice of collapsing the fault at the input or at the output is not significant, whereas for the other 4 basic gates, it's always more advantageous to collapse the faults at the gate input because more faults can be collapsed. It is even more advantageous if we collapse the stuck-at faults for inverters and buffers in the same way as the other basic gates because of the rippling effect of collapsing through a gate network. These effects are shown in Figure 3.3.

3.3 Gate-Delay Faults

Some physical defects cause a gate or wire to respond more slowly than expected. There are two types of faults in this so-called gate delay fault model, namely *slow-to-rise* (SR) and *slow-to-fall* (SF) [32]. If the response is too slow when the signal changes from logical 0 to 1 (rising), then it is a SR fault. On the other hand, if the signal is too slow when it changes from 1 to 0 (falling), it is a SF fault. For a signal

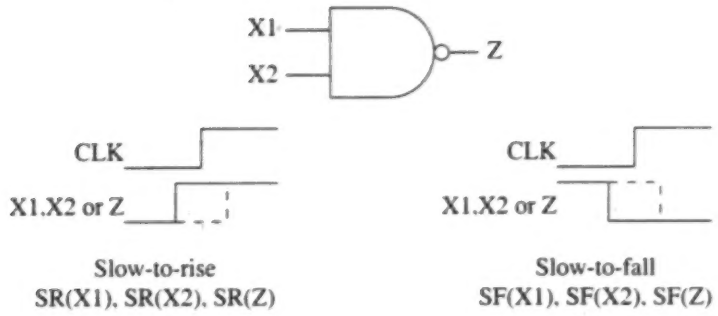


Figure 3.4: Gate-Delay Faults Associated with 2-input NAND Gates

change to be significantly slow to cause a fault it must be slow enough to miss the correct signal change before the arrival of the next clock signal. We assume that the circuit under consideration is synchronous although we will only consider the effects of physical defects in combinational logic.

There is another popular fault model, called the path delay fault model, which also deals with signals that change more slowly than expected. However, this model assumes there are a number of gates which are slower than normal gates, yet their effect is only significant when a signal propagates through all these gates; in other words, the effect is only significant when numerous slow responses are accumulated. The path delay fault model is useful for modeling the effects of silicon process variations that affect many gates only slightly. Since there are very many possible different signal propagation paths in a circuit, it is difficult to simulate all of these paths for path delay faults. As a result, this fault model was not implemented in this research.

Two consecutive test patterns are required to trigger a gate-delay fault. In order to trigger a slow-to-rise fault, the first test pattern has to set the wire in question to logical 0. Then the second test pattern must attempt to set it back to a logical 1. If the wire is slow-to-rise, then it will stay at a logical 0 when the next clock cycle starts. If this fault effect propagates to any of the observing points, then the fault is detected. Similarly, slow-to-fall faults can be triggered by setting the wire to 1 with the first pattern, and then to 0 with the second pattern.

Each wire in the circuit can be either slow-to-rise or slow-to-fall, therefore the number of gate-delay faults in the circuit equals twice the number of wires before fault collapsing. The fault collapsing rules for the gate-delay fault model are different

NAND (input X1 X2, output Z)

Fault	First X1X2	Second X1X2	Good ZZ^+	Faulty ZZ^+
SR(X1)	0x	11	10	11
SF(X1)	1x	01	x1	x0
SR(X2)	x0	11	10	11
SF(X2)	x1	10	x1	x0
SR(Z)	11	(!1)	01	00
SF(Z)	(!1)	11	10	11

(!1) = not all ones

Table 3.2: Triggering Conditions of Stuck-At Faults for 2-input NAND Gate

from those of the stuck-at fault model. Let's take a 2-input NAND gate as an example (See Figure 3.4). Table 3.2 lists the triggering conditions for all the gate delay faults associated with a 2-input NAND gate. In this table, Z denotes the output signal for the first input vector while Z^+ denotes the same output signal for the second input vector. The lower case "x" denotes a don't care value. The table entries imply that there are no two faults that have exactly the same triggering condition; however, any test pattern that detects the SF(Z) fault (slow-to-fall on output signal Z) also detects either SR(X1) or SR(X2). In other words, when a SF(Z) fault is detected, we can't be sure whether it is really a SF(Z) or a SR at one of the inputs. We cannot collapse the SR faults at the inputs because the detection of SF(Z) does not guarantee the detection of all the SR faults at the inputs, whereas detecting any of the SR at the inputs would guarantee the detection of SF(Z). As a result, the SF(Z) fault can be collapsed, and the NAND gate is left with only 5 gate-delay faults. Similar fault collapsing rules can be used to reduce the number of gate delay faults from the other standard logic gates. Triggering conditions for the other basic logic gates can be found in Tables 3.5 to 3.10.

Note that again, for inverters and buffers as with stuck-at faults, we have the option of collapsing the delay faults at either an input or the output. In order to have the minimum number of faults after collapsing, we should choose the same collapsing direction for all the gate-delay faults. Figure 3.5 shows the advantage of collapsing the faults in one direction. We should also note that the preferred collapsing direction

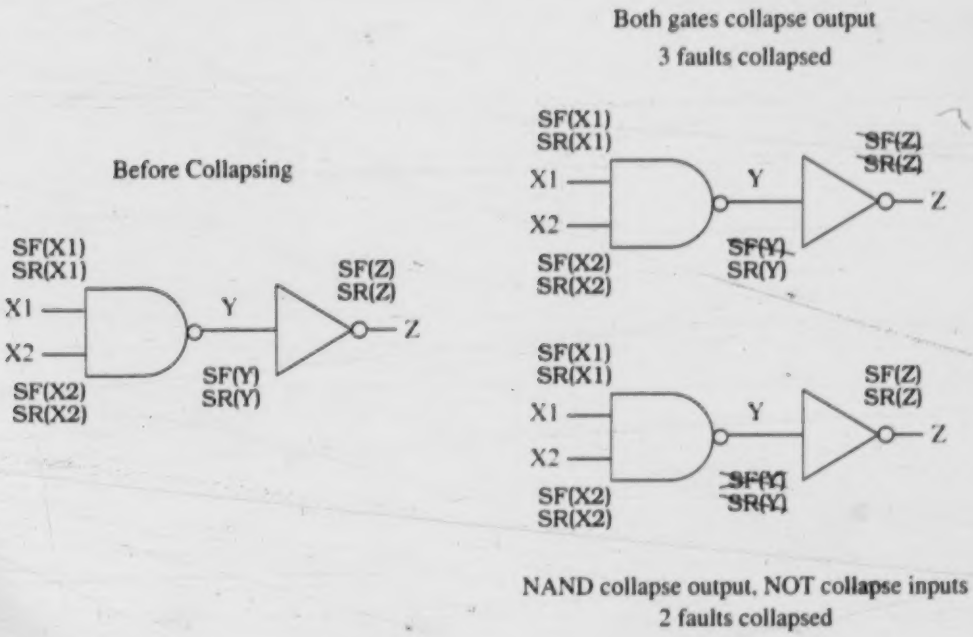


Figure 3.5: Fault Collapsing for the Gate-Delay Fault Model

for the gate-delay fault model (collapsing the outputs) is different from that of the stuck-at fault model (collapsing the inputs). This is not significant if we do not take advantage of the relationships between different fault models, which will be discussed in Section 3.5.

3.4 Stuck-Open Faults

The third fault model used in this research is the *CMOS transistor stuck-open* fault model. This fault model is specific to CMOS technology, whereas the two earlier fault models are technology independent. The CMOS transistor stuck-open fault model, often simply called the *stuck-open* model, targets problems that cause the source to drain channel within CMOS transistors to be permanently opened. In CMOS technology, transistors come in two types, namely the N-transistor and P-transistor. When an N-transistor channel is open circuited, the fault is called an N-transistor open fault (NO); on the other hand, an open circuited P-transistor is called a P-transistor open fault (PO).

In CMOS technology, P-transistors are responsible for driving a gate output signal

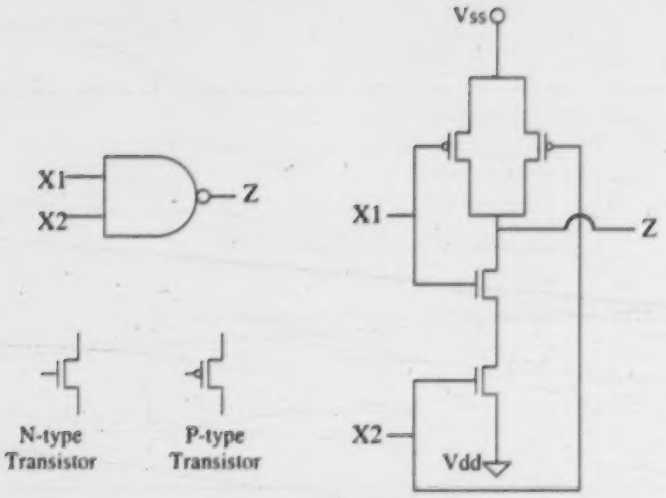


Figure 3.6: A 2-input NAND Gate with Stuck-Open Fault

from low to high, while N-transistors are used for driving signals from high to low. See Figure 3.6 for the schematic of a CMOS 2-input NAND gate. If a P-transistor channel in a logic gate is open circuited, then for some input vector sequences, the output signal will stay low erroneously.

Since stuck-open faults are associated with transistors, before fault collapsing, the number of stuck-open faults in a circuit equals the number of transistors in the circuit. For NOT, NOR and NAND gates, the number of transistors for a logic gate is equal to twice the number of gate inputs. AND gates, OR gates and buffers are special cases since these components can have different possible implementations in a CMOS environment. In this research, these gates are assumed to be constructed from the primitive NOT, NOR and NAND gates.

Table 3.3 lists the triggering conditions for all the stuck-open faults of a 2-input NAND gate. As shown in Figure 3.6, the N-transistors of the NAND gate are connected in series while the P-transistors are connected in parallel. Since the N-transistors are in series, if any of the N-transistor fails, the output signal cannot be driven low under any circumstances. From the testing point of view, all we know is that there is a failing N-transistor in the gate but we cannot know which one is actually failing. Therefore, the NO faults of a NAND gate are equivalent and can be collapsed into a single NO fault. The same holds true for any series connection of

NAND (input X1 X2, output Z)

Fault	First X1X2	Second X1X2	Good ZZ^+	Faulty ZZ^+
NO(X1)	(!1)	11	10	11
PO(X1)	11	01	01	00
NO(X2)	(!1)	11	10	11
PO(X2)	11	10	01	00

(!1) = not all ones

Table 3.3: Triggering Conditions for Gate-Delay Faults Affecting a 2-input NAND Gate

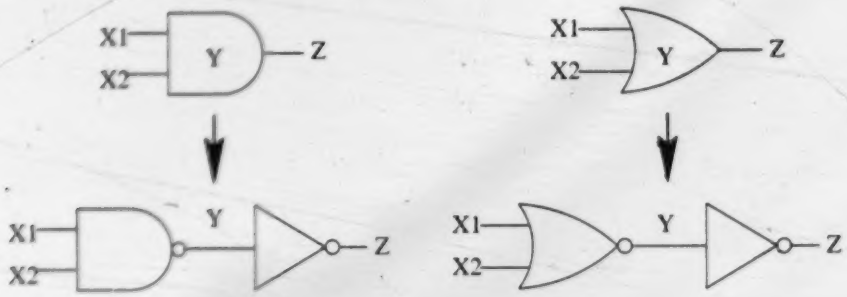


Figure 3.7: Common Structure of CMOS AND Gates and OR Gates

transistors in any logic gates. The fault tables for NOT and NOR gates are shown in Tables 3.5 and 3.9, respectively.

As mentioned above, AND gates, OR gates and buffers are assumed to be constructed using NOT, NOR and NAND gates (see Figure 3.7). The fault table for a 2-input AND gate is as shown in Table 3.4. The Y columns of the table indicate the intermediate signal between the NAND gate and the NOT gate. Thus NO(Y) and PO(Y) are called the *internal stuck-open faults*. As seen in the table, the triggering conditions for NO(X1), NO(X2) and PO(Y) are identical (except for the Y columns). The three faults are therefore equivalent, and can be collapsed into a single fault. Also note that the triggering condition for PO(Y) is similar to SR(Z), while NO(Y) is similar to SF(Z). Identical triggering conditions imply equivalent faults. However, since they are faults in different fault models, collapsing them would result in an incorrect fault coverage calculation[1] for at least some of the fault types. Fault

[1]The common way of calculating fault coverage is to divide the number of faults detected by the

AND (input X1 X2, intermediate Y, output Z)

Fault	First X1X2	Second X1X2	Good Y	Good ZZ+	Faulty ZZ+
NO(X1)	(!1)	11	10	01	00
PO(X1)	11	01	01	10	11
NO(X2)	(!1)	11	10	01	00
PO(X2)	11	10	01	10	11
NO(Y)	11	(!1)	01	10	11
PO(Y)	(!1)	11	10	01	00

(!1) = not all ones

Table 3.4: Triggering Conditions for Stuck-Open Faults Affecting 2-input AND Gate

equivalence among fault models will be discussed in the next section. Similar fault collapsing rules are used for OR gates and buffers. The fault tables for the OR gate is listed in Table 3.10. The fault table for the buffer is listed in Table 3.6.

3.5 Fault Implication and Equivalence Among Fault Models

In the previous three sections, the fault models used in this research and their fault collapsing rules were introduced. When using these three models together in a fault simulation, the fault simulation algorithm can be made much more efficient by considering *fault implications* and *fault equivalence* among the fault models. Recognizing and exploiting these relationships reduces the amount of effort required in fault simulation.

Fault implication is the one-way relationship between faults where if a fault M is detected, then fault N must be detected, where the opposite is not necessarily true. In this case, we say that fault M *implies* fault N. For example, in order to trigger a slow-to-rise gate-delay fault on the input of an inverter, a first test pattern has to force the input to logic 0, and a second pattern must force the input to logic 1. On the other hand, to trigger a stuck-at 0 fault on the input of an inverter, a test pattern has to force the input to logic 1. Since both faults require a test pattern to force the input of the inverter to 1, if the slow-to-rise fault is detected, it must be

total number of faults after collapsing.

true that the stuck-at 0 fault is also detected. In other words, the slow-to-rise fault implies the stuck-at 0 fault. The reverse implication is not true because detection of the delay fault requires a stronger condition than the stuck-at fault (2 patterns versus 1). Figure 3.8 shows all the one-way relationships that exist between the fault models used in this research. In the figure, a fault at an arrow tail implies the fault at the same arrow's head.

When two faults of the same fault model type are equivalent, they are collapsed into one fault of that common type. When two different faults from different models are equivalent, however, you cannot collapse any of them because this would result in an incorrect fault coverage calculation. One way to exploit equivalence relationships without collapsing the faults is to view the relationship as two separate fault implications. For example, the slow-to-fall fault on the output of a NAND gate is equivalent to the N-transistor open fault on its input (we can see that by comparing the triggering condition of the faults). Instead of collapsing these faults into a single fault, they are kept as two separate faults while having them separately imply each other. When a test sequence detects either one of them, the other fault can be considered detected using the implication relationship, without requiring further simulation.

The fault list used in our fault simulators supports both fault implication and fault equivalence, as discussed in Section 5.2.2. The effect of these relationships is also presented there.

NOT Gate (input X, output Z)

Fault	First X	Second X	Good ZZ^+	Faulty ZZ^+	Collapsing Faults
SA0(X)	x	1	x0	x1	SA1(Z)
SA1(X)	x	0	x1	x0	SA0(Z)
SA0(Z)	x	0	x1	x0	
SA1(Z)	x	1	x0	x1	
SR(X)	0	1	10	11	
SF(X)	1	0	01	00	
SR(Z)	1	0	01	00	SF(X)
SF(Z)	0	1	10	11	SR(X)
NO(X)	0	1	10	11	
PO(X)	1	0	01	00	

Table 3.5: Triggering Conditions for Faults Affecting the NOT Gate

Buffer (input X, output Z)

Fault	First X	Second X	Good ZZ^+	Faulty ZZ^+	Collapsing Faults
SA0(X)	x	1	x1	x0	SA0(Z)
SA1(X)	x	0	x0	x1	SA1(Z)
SA0(Z)	x	1	x1	x0	
SA1(Z)	x	0	x0	x1	
SR(X)	0	1	01	00	
SF(X)	1	0	10	11	
SR(Z)	0	1	01	00	SR(X)
SF(Z)	1	0	10	11	SF(X)
NO(X)	0	1	01	00	
PO(X)	1	0	10	11	

Table 3.6: Triggering Conditions for Faults Affecting the BUFF Gate

NAND Gate (input X1 X2, output Z)

Fault	First X1X2	Second X1X2	Good ZZ^+	Faulty ZZ^+	Collapsing Faults
SA0(X1)	xx	11	x0	x1	SA1(Z)
SA1(X1)	xx	01	x1	x0	
SA0(X2)	xx	11	x0	x1	SA1(Z)
SA1(X2)	xx	10	x1	x0	
SA0(Z)	xx	(!1)	x1	x0	
SA1(Z)	xx	11	x0	x1	
SR(X1)	0x	11	10	11	
SF(X1)	1x	01	x1	x0	
SR(X2)	x0	11	10	11	
SF(X2)	x1	10	x1	x0	
SR(Z)	11	(!1)	01	00	
SF(Z)	(!1)	11	10	11	SR(X1),SR(X2)
NO(X1)	(!1)	11	10	11	
PO(X1)	11	01	01	00	
NO(X2)	(!1)	11	10	11	NO(X1)
PO(X2)	11	10	01	00	

(!1) = not all ones

Table 3.7: Triggering Condition for Faults Affecting the NAND Gate

AND Gate (input X1 X2, output Z)

Fault	First X1X2	Second X1X2	Good ZZ^+	Faulty ZZ^+	Collapsing Faults
SA0(X1)	xx	11	x1	x0	SA0(Z)
SA1(X1)	xx	01	x0	x1	
SA0(X2)	xx	11	x1	x0	SA0(Z)
SA1(X2)	xx	10	x0	x1	
SA0(Z)	xx	11	x1	x0	
SA1(Z)	xx	(!1)	x0	x1	
SR(X1)	0x	11	01	00	
SF(X1)	1x	01	x0	x1	
SR(X2)	x0	11	01	00	
SF(X2)	1x	10	x0	x1	
SR(Z)	(!1)	11	01	00	SR(X1),SR(X2)
SF(Z)	11	(!1)	10	11	
NO(X1)	(!1)	11	01	00	
PO(X1)	11	01	10	11	
NO(X2)	(!1)	11	01	00	NO(X1)
PO(X2)	11	10	10	11	
NO(Y)	11	(!1)	10	11	
PO(Y)	(!1)	11	01	00	NO(X1)

(!1) = not all ones

Table 3.8: Triggering Conditions for Faults Affecting the AND Gate

NOR Gate (input X1 X2, output Z)

Fault	First X1X2	Second X1X2	Good ZZ^+	Faulty ZZ^+	Collapsing Faults
SA0(X1)	xx	10	x0	x1	
SA1(X1)	xx	00	x1	x0	SA0(Z)
SA0(X2)	xx	01	x0	x1	
SA1(X2)	xx	00	x1	x0	SA0(Z)
SA0(Z)	xx	00	x1	x0	
SA1(Z)	xx	(!0)	x0	x1	
SR(X1)	0x	10	x0	x1	
SF(X1)	1x	00	01	00	
SR(X2)	x0	01	x0	x1	
SF(X2)	x1	00	01	00	
SR(Z)	(!0)	00	01	00	SF(X1),SF(X2)
SF(Z)	00	(!0)	10	11	
PO(X1)	(!0)	00	01	00	
NO(X1)	00	10	10	11	
PO(X2)	(!0)	00	01	00	PO(X1)
NO(X2)	00	01	10	11	

(!0) = not all zeros

Table 3.9: Triggering Conditions for Faults Affecting the NOR Gate

OR Gate (input X1 X2, output Z)

Fault	First X1X2	Second X1X2	Good ZZ^+	Faulty ZZ^+	Collapsing Faults
SA0(X1)	xx	10	x1	x0	
SA1(X1)	xx	00	x0	x1	SA1(Z)
SA0(X2)	xx	01	x1	x0	
SA1(X2)	xx	00	x0	x1	SA1(Z)
SA0(Z)	xx	(!0)	x1	x0	
SA1(Z)	xx	00	x0	x1	
SR(X1)	0x	10	x1	x0	
SF(X1)	1x	00	10	11	
SR(X2)	x0	01	x1	x0	
SF(X2)	x1	00	10	11	
SR(Z)	00	(!0)	01	00	
SF(Z)	(!0)	00	10	11	SF(X1).SF(X2)
PO(X1)	(!0)	00	10	11	
NO(X1)	00	10	01	00	
PO(X2)	(!0)	00	10	11	PO(X1)
NO(X2)	00	01	01	00	
PO(Y)	00	(!0)	01	00	
NO(Y)'	(!0)	00	10	11	PO(X1)

(!0) = not all zeros

Table 3.10: Triggering Conditions for Faults Affecting the OR Gate

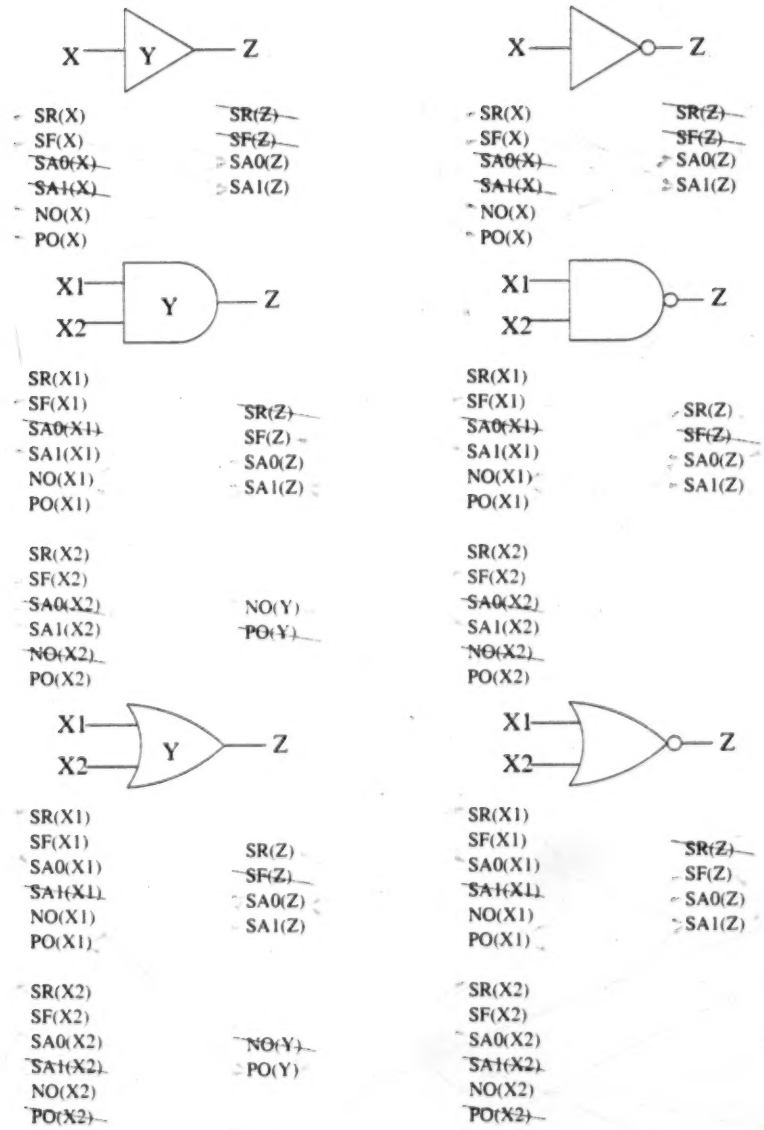


Figure 3.8: Fault Implication and Fault Collapsing Relationships

Chapter 4

Fault Simulation

4.1 Introduction

When integrated circuits are manufactured, physical defects are likely to be introduced because of the extremely small dimensions that are involved. Boolean fault models are used to represent these physical defects, as the previous chapter has described. After manufacturing, it is necessary to distinguish the faulty circuits from the good ones through testing. To detect faults in a circuit under test (CUT), a sequence of test patterns has to be applied to the inputs of the CUT. The outputs of the CUT are then recorded and compared to the sequence of expected outputs from a good circuit. If the output sequences differ, then a faulty circuit has been detected.

There are generally two methods of circuit testing. The traditional one is to test a circuit with external automatic test equipment (ATE). Modern ATE tends to be very expensive and its lifetime tends to be relatively short before it becomes obsolete. An alternative way is to use *Built-In Self-Test* (BIST). BIST is a kind of

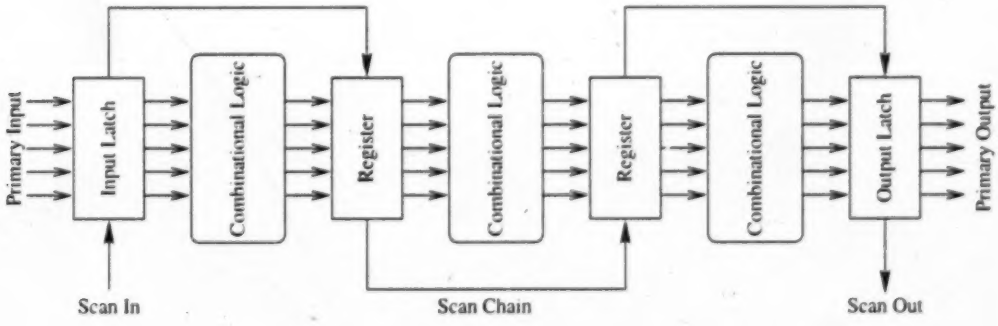


Figure 4.1: The Scan Chain/BIST Architectures

circuit architecture where the testing hardware is included as part of the circuit (See Figure 4.1). Most circuits today are *sequential circuits*, which have memory elements such as flip-flops in their design. Many popular BIST architectures, such as the LOCST architecture [21], modify the memory elements into special latches that can operate using either the system clock or a testing clock, depending on the selected mode of operation. The latches are further connected together as one or more scan chains so that data can be shifted serially in to and out of the latches during testing. This feature not only allows the state of the latches to be observed, it also allows the sequential circuit to be partitioned into several accessible combinational blocks. A *combinational circuit* is one that does not have any internal memory elements. Testing such a circuit block is generally easier than a sequential circuit because the outputs of a combinational block depend only on the present inputs. Thus, if a test pattern is supplied as input to a combinational block, the output of the block can be compared directly with that of a fault-free block without considering any other conditions (such as the internal memory states in a sequential circuit). Simulation of a combinational circuit is often enough to evaluate many important differences between alternative implementations of fault simulation algorithms. All the fault simulators implemented in this research handle only combinational circuits.

Another important characteristic of BIST is the use of pseudo-Random Test Pattern Generators (RTPGs). In order to test a circuit, a sequence of test patterns has to be supplied as inputs to the circuit under test. In a BIST design, the hardware to generate the test sequence is included as part of the circuit. The input test sequence could be generated deterministically by analyzing the structure of the circuit itself. When using a BIST design, these test patterns have to be stored in ROMs (Read only memory) as part of the circuit. Even if there are relatively few deterministic test patterns, the hardware requirement for pattern storage is still relatively higher than that required typically for RTPG. RTPGs are special counters that count in pseudo-random order. The size of a RTPG grows only very slowly (usually logarithmically) with the length of the self-test, thus it is often economical to include the RTPG hardware into on-chip BIST circuits.

There are many different RTPGs, and most of the RTPGs can be further customized with different parameters. Some well known RTPGs are the Linear Feedback

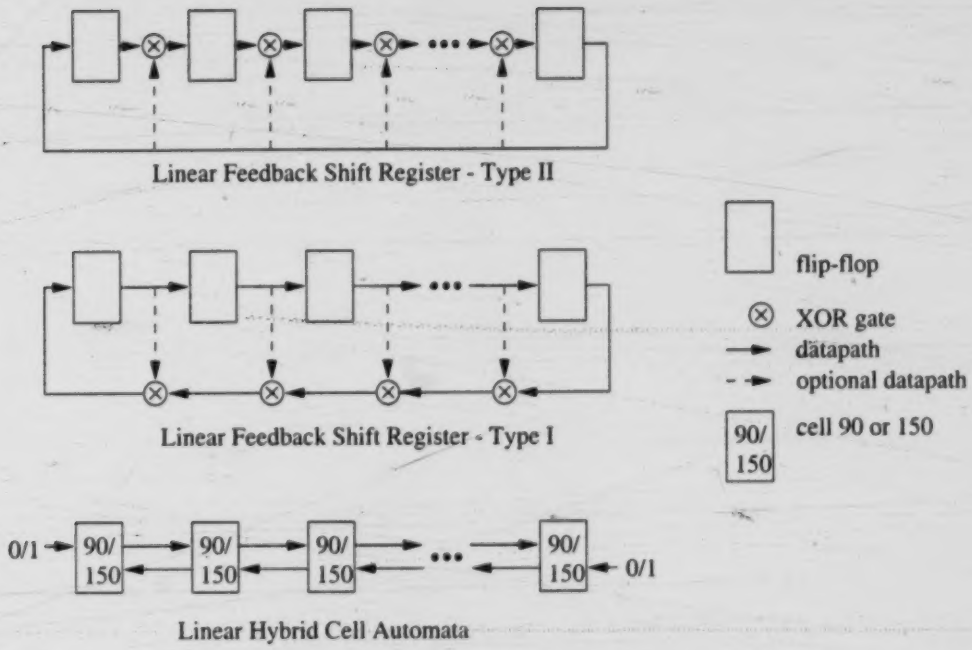


Figure 4.2: Hardware Configuration of RTPGs

Shift Register (LFSR) and the Linear Hybrid Cellular Automata (LHCA) [4]. Figure 4.2 illustrates the structure of these RTPGs. An LFSR implements polynomial division. The output sequence of an LFSR can be customized by choosing different polynomial divisors. An LHCA is constructed by connecting together two types of simple, one-bit cellular automata in a linear array. One type of automaton (often called rule 90) calculates its next value as the XOR of the value of its two neighbors, while the other type (often called rule 150) takes the XOR of its neighbors and its own value. Putting together these automata as a linear array with different configurations and different initial values can result in different sequences. The random patterns generated by an LFSR and LHCA can be further modified using a Maximum Transition Counter (MTC) [6]. A MTC is a sequential circuit with maximum length state sequence whose adjacent states different in all bits except one. The hybrid method mixes the test sequence generated by the RTPG and the MTC to form a new RTPG, which generally produces test sequences that appear to detect sequential faults more easily, at the cost of more hardware. Figure 4.3 illustrates the different fault coverages obtained by feeding different pseudo-random test patterns into the same circuit.

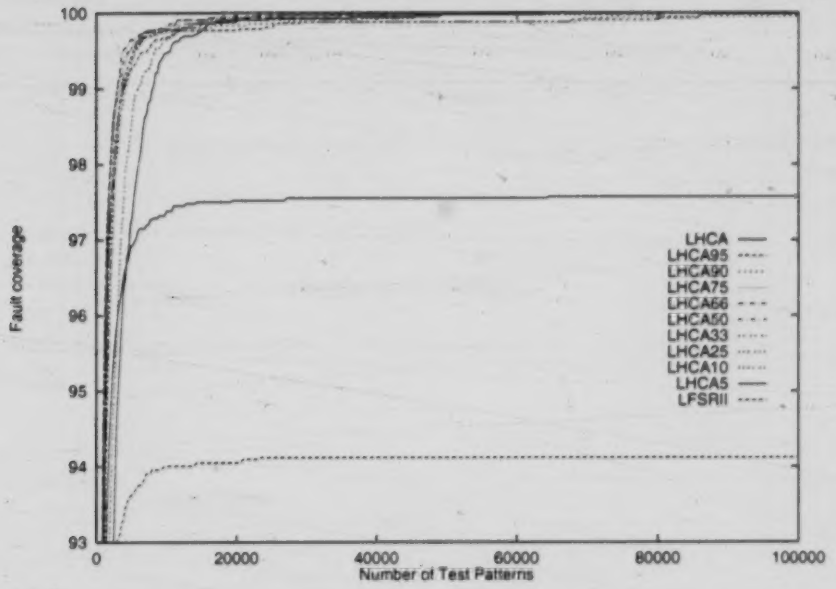


Figure 4.3: Fault Coverage of Different RTPGs

Circuit manufacturers are interested in finding RTPGs that are the most effective at detecting the expected faults in their particular circuit with the least possible hardware overhead. With so many options, an efficient tool is needed to rapidly evaluate the effectiveness for a given circuit of the many possible alternative RTPGs.

As explained in Chapter 1, fault simulation is the process of simulating the operation of a circuit with the presence of faults to evaluate the effectiveness of a proposed test. To evaluate a test sequence, the patterns have to be applied as inputs to the simulated CUT. The number of test patterns required for detecting the presence of all the faults of interest is recorded, i.e. the test length. This number can be used directly as a measure of the efficiency of the test sequence corresponding to the CUT. The shorter the test, the less time is required using expensive ATE for the CUT to be fully tested. When using RTPGs for generating test sequences, sometimes it is acceptable to detect fewer than 100% of the faults of interest. The remaining undetected faults can be handled by applying extra test patterns to the CUT. This method is useful when there are known to be hard-to-detect faults in the circuit, and an RTPG can detect all the rest of the faults within a relatively short period of time. A significant amount of testing time can be saved if these hard-to-detect faults are

treated as special cases, with individually stored patterns. Therefore, fault simulation is used not only to find out the minimum length for a test sequence, but also the less than complete fault coverage (i.e. the percentage of faults being detected) for a limited number of test patterns. It also indirectly finds difficult-to-detect or likely impossible-to-detect faults. Note that in large circuits, fault simulation alone cannot prove that a fault is truly impossible to detect.

4.2 Sequential Fault Simulation

The most basic fault simulation algorithm is the *sequential fault simulation algorithm.* In this algorithm, a list of faults of interest is first constructed from the structure of the CUT. The fault list is usually then reduced by exploiting fault equivalences and fault implications. A test pattern from the test sequence is then applied to a simulated fault-free circuit. The logical values at the outputs of this fault-free circuit are recorded. Then, the first fault from the fault list is inserted into the model of the CUT, and the circuit output sequence in the presence of this fault is re-computed. If the output of this fault simulation is different from that of a fault-free simulation, then the fault is considered detected and is removed from the fault list; otherwise, the fault list will remain unchanged. The next fault from the fault list is then inserted into the fault-free circuit and the effect of the same test pattern in the presence of the new fault is re-computed. This process continues until all the faults in the fault list have been tested. At this time, simulation of the first test pattern is finished. The next test pattern can now be loaded and the circuit simulation process repeats. This whole simulation process continues until either the fault list is empty or we run out of test patterns. If sequential fault models are used for representing physical defects, then the logical states of all circuit nodes in the fault-free circuit for a test pattern have to be recorded for the next test pattern because they will need to be referenced.

As seen in Figure 4.4, this fault simulation algorithm has two levels of nested looping: an outer loop that loads individual test patterns sequentially (looping through time), and an inner loop that loads faults from the fault list one after another (looping through space). Each loop is essentially advancing through an array of data defined by the test sequence and the fault list. The two arrays can be viewed as defining a

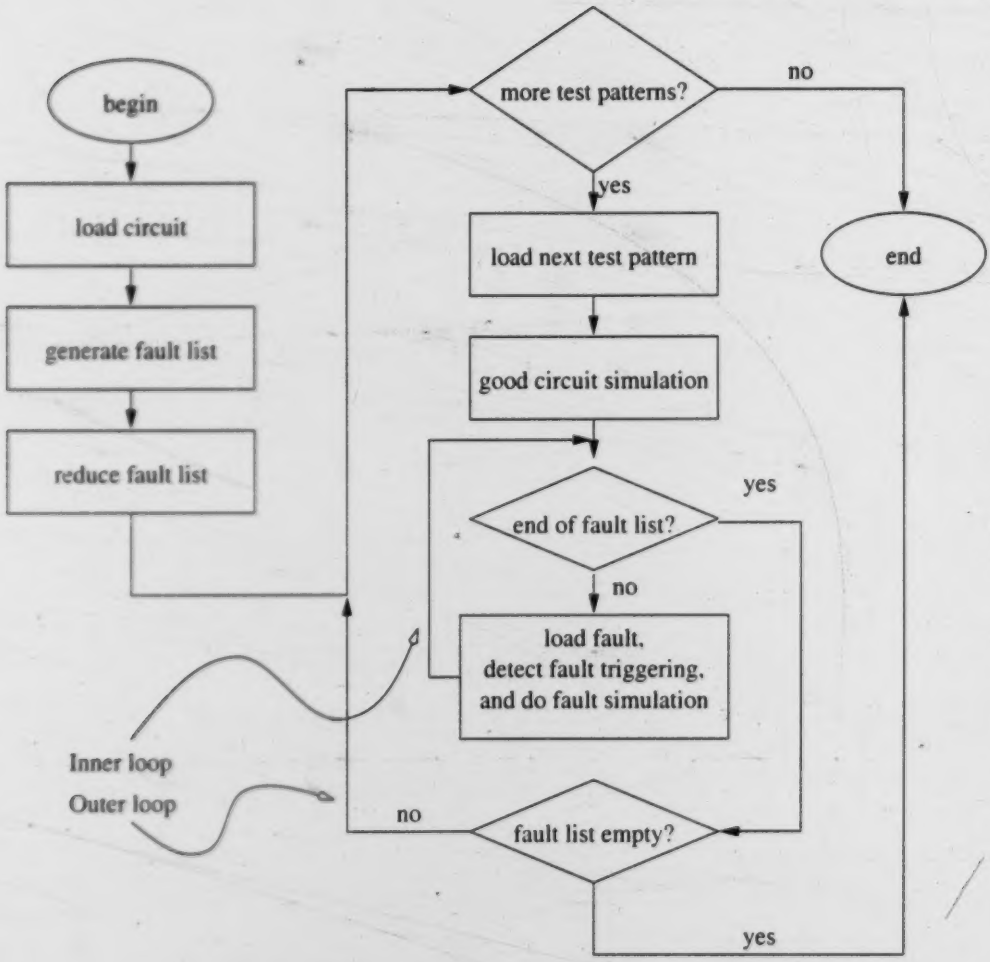


Figure 4.4: Flow Chart of the Basic Fault Simulation Algorithm

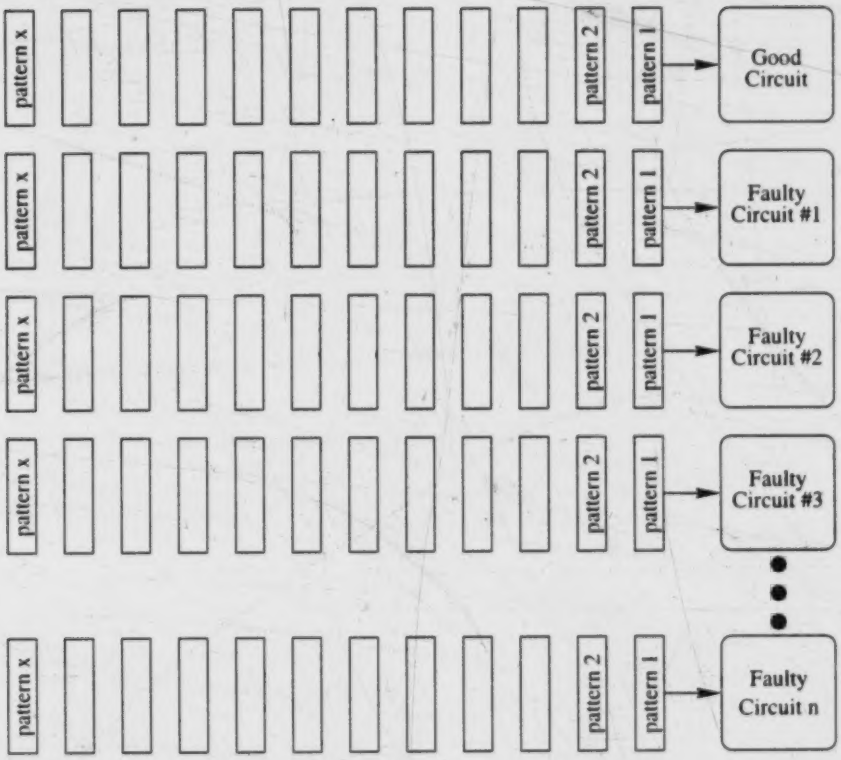


Figure 4.5: 2-Dimensional Fault Simulation Space

two-dimensional fault simulation space as shown in Figure 4.5. The pattern dimension can be viewed as a time dimensions, and the fault dimension can be viewed as a space dimension. It is possible to modify this algorithm to speed up the processing by running multiple simulations in parallel. The degree of parallelism in the pattern space, i.e. the number of test patterns simulated in parallel, will be denoted by p; whereas the degree of parallelism in the fault space, i.e. the number of faults simulated in parallel, will be denoted f. An algorithm with $p = n$ means that it simulates n patterns in parallel. A sequential algorithm thus has both p and f set equal to 1.

Recalling our objectives from Chapter 1, we are going to describe the implementations of three different fault simulation algorithms on C•RAM. The algorithm that runs multiple faulty circuits in parallel ($p = 1, f > 1$) is called the *fault-parallel simulation algorithm* [28]. *Pattern-parallel simulation*, specifically the Parallel Pattern Single Fault Propagation (PPSFP) algorithm [31], simulates multiple test patterns

in parallel ($p > 1, f = 1$). Finally, hybrid algorithms can be defined for parallelizing simultaneously in both the fault and test pattern dimensions ($p >= 1, f >= 1$). One adaptive hybrid algorithm is called the Dynamic Two-Dimensional Parallel Simulation [18].

Other algorithms using massively parallel SIMD machines to speed up fault simulation have been designed and evaluated by other researchers. In [23], two algorithms were proposed, namely Gate Parallel Level Serial Simulation (GPLSS) and Parallel Pattern-Parallel Fault Simulation (PPPFS). These algorithms were implemented on Thinking Machines Corporation's Connection Machine. The results presented in [23] were given in terms of the actual simulation run time on real hardware, which makes it very difficult to compare their performance directly with the results obtained in this research using emulated C•RAM hardware.

4.3 Fault-Parallel Fault Simulation

The first parallel algorithm, Fault-Parallel Fault Simulation, evaluates in parallel multiple faulty circuits ($p = 1, f > 1$), i.e. each PE simulates a different faulty circuit. Each faulty circuit is inserted with a different fault in the fault list. A single test pattern is input to the CUT, and corresponding gates in each circuit are evaluated in parallel. The output of each of the faulty circuits is compared with the corresponding output of the fault-free circuit. If an output of a faulty circuit is different from that of the fault-free one, then the fault is marked as detected. After all gate evaluations have been performed for the particular test pattern, the marked faults are removed from the fault list. The parallelism in our 2-D space is demonstrated in Figure 4.6. Notice how sets of patterns have been looped in the figure to identify identical patterns that are fault simulated together in parallel in the same session.

One potential problem of this algorithm is the size of the fault list. If, at the beginning of the simulation, the number of faults in the fault list is greater than or equal to the number of PEs available, then more than one pass is required for the complete evaluation of a single test pattern. (In Figure 4.6 only one pass is assumed to be sufficient.) Problems also arise when the fault list is too small. For example, if n PEs are available, and there is only one fault in the fault list, then the simulator has to

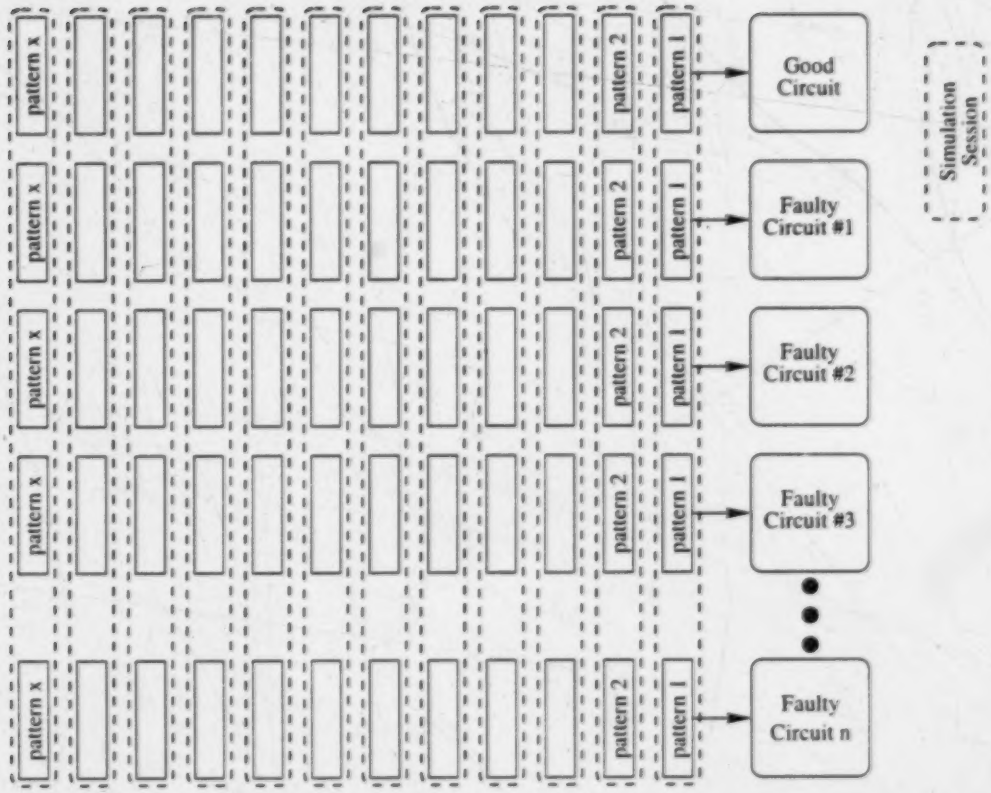


Figure 4.6: 2-D Fault Simulation Space of Fault-Parallel Fault Simulation

run with only one PE active. This inefficiency can be solved by running more than one test pattern in parallel, which suggests the other fault simulation algorithm: Hybrid Parallel Fault Simulation. Other problems with Fault-Parallel Fault Simulation will be discussed in Chapter 7.

4.4 Pattern-Parallel Fault Simulation

The Pattern-Parallel Fault Simulation algorithm is basically the Parallel Pattern Single Fault Propagation (PPSFP) [31] algorithm extended to fit onto the SIMD architecture. In this algorithm, a number of test patterns are simulated in parallel ($p > 1, f = 1$), whereas in reality, these patterns will be applied to a real CUT one after another. Since only combinational circuits are considered in this research, the output of a circuit depends only on its current input, thus simulating test pat-

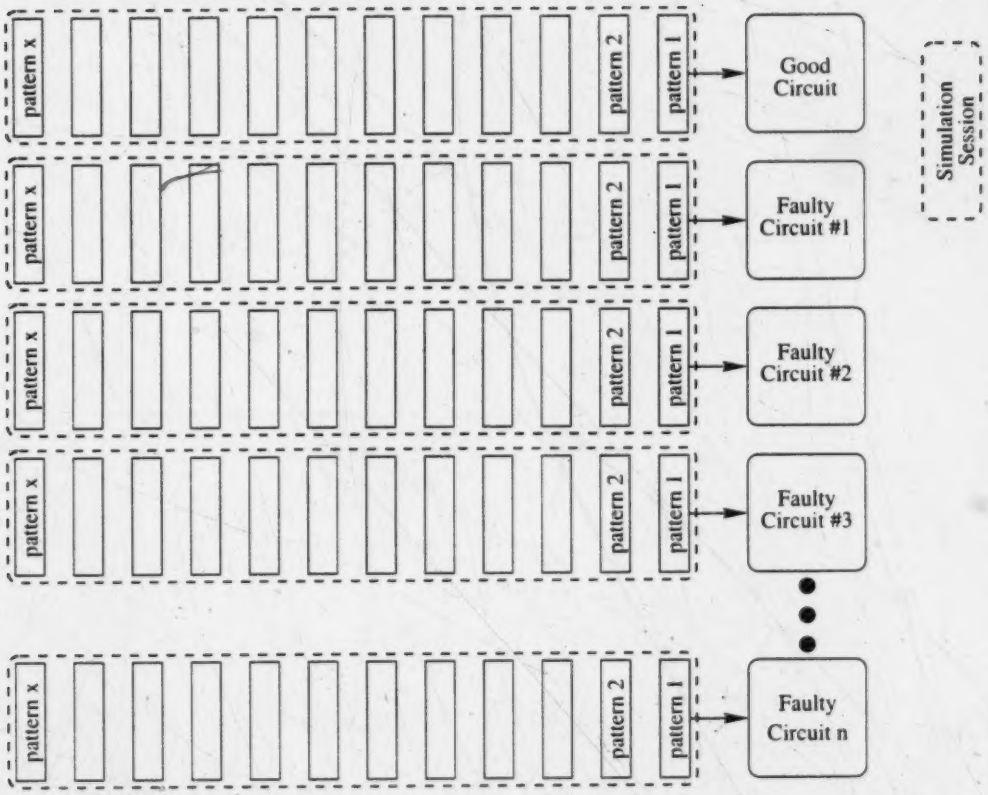


Figure 4.7: 2-D Fault Simulation Space of Pattern-Parallel Fault Simulation

terns in parallel does not cause any problems, except when simulating sequential faults. Sequential faults, as described in Chapter 3, introduce sequential behaviours into combinational circuits. The handling of sequential faults in this algorithm is the same as the approach used in the PPSFP algorithm, which will be described in Chapter 5.

The pattern-parallel algorithm first simulates the fault-free circuit as it responds to the group of test patterns. Then the first fault in the fault list is inserted and its effect is simulated for all patterns in the session in parallel. The remaining faults in the fault list are simulated one after another in the same manner. Figure 4.7 shows the parallelism achieved in the 2-D problem space.

The conventional fault simulator implemented as a benchmark for this thesis used the PPSFP algorithm. In SIMD programming, it is often true that local enhancements

that work for a sequential algorithm[1] do not always improve the parallel version. Since the PPSFP algorithm on a conventional machine is very similar to the Pattern-Parallel Fault Simulation algorithm on C•RAM, the two implementations will be compared closely in Chapter 6.

4.5 Hybrid Parallel Fault Simulation

When a fault simulator parallelizes fault simulation in both the pattern space and the fault space, it is called a hybrid parallel fault simulator. Basically there are two ways to implement such an algorithm. One way is to simulate the same amount of faults and test patterns in parallel throughout the program ($p = n, f = m$, where $n > 1$ and $m > 1$ are fixed values). Since the degree of parallelism is static, it is called a *static hybrid parallel fault simulation*. Another method which determines a different p and f at each round of the simulation is called a *dynamic hybrid parallel fault simulation* ($p = n, f = m$, where n and m vary).

In [18], the Dynamic Two-Dimensional Parallel Simulation (DTDPS) technique is presented. It was observed that the purely fault-parallel fault simulation technique works most efficiently when there are many relatively easy-to-detect faults in the fault list, while pure pattern-parallel fault simulation techniques are most efficient with a small number of relatively hard-to-detect faults with many test patterns to be simulated. The authors attempted to take advantage of the two algorithms by dynamically using fault-parallel fault simulation at the beginning of the simulation, switching increasingly to pattern-parallel fault simulation towards the end of the whole simulation process. The technique was implemented on the FACOM vector processor.

Figure 4.8 shows the 2-D fault simulation space of hybrid parallel fault simulation. At the beginning of the simulation, pure Fault-Parallel Fault Simulation is used as described in Section 4.3. After the simulation of a test pattern is finished, the percentage of faults that were detected by that test pattern (fault coverage) is evaluated. If the fault coverage exceeds a certain percentage FC_H, the next pass will simulate two test patterns in parallel. For example, if n PEs are available for simulation, then

[1] An algorithm that runs without SIMD hardware.

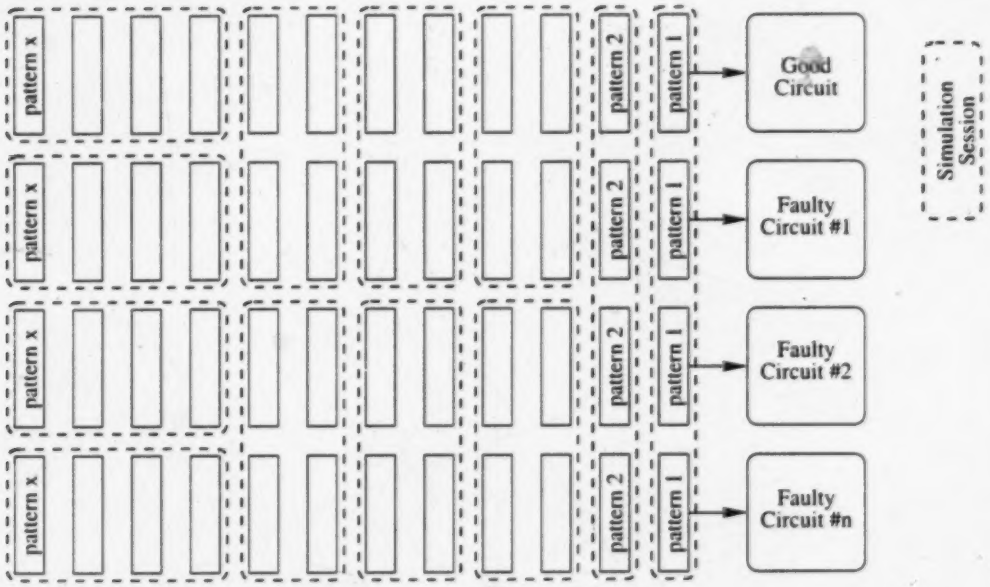


Figure 4.8: 2-D Fault Simulation Space of Hybrid Parallel Fault Simulation

in the first pass, n PEs are used for fault-parallel fault simulation of the first test pattern. After the first pass, if say more than $FC_H = 25\%$ of the faults are detected, then in the next pass, two patterns are simulated in parallel. In this case, $n/2$ PEs are allocated for each test pattern. As the fault coverage again exceeds FC_H, the number of test patterns simulated in parallel also increases. After a certain point, when there are only a few faults left in the fault list, the simulation can switch to purely pattern-parallel fault simulation. The implementation details are discussed in Chapter 8.

4.6 Evaluation of Fault Simulators

4.6.1 Faults, Undetected Faults, and Triggered Faults

To evaluate the fault simulation algorithms described above, we have to investigate the process of detecting a fault in more detail. Each undetected fault can be either *triggered* or *not-triggered* by each test pattern. If the fault is triggered, it will be either *detected* or *undetected*. These possibilities are shown in Figure 4.9. The total number of undetected faults decreases monotonically as a function of time in simulation (or

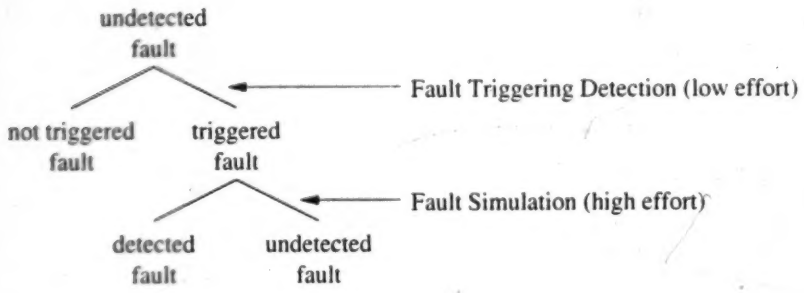


Figure 4.9: Classes of Faults with Respect to Fault Detection

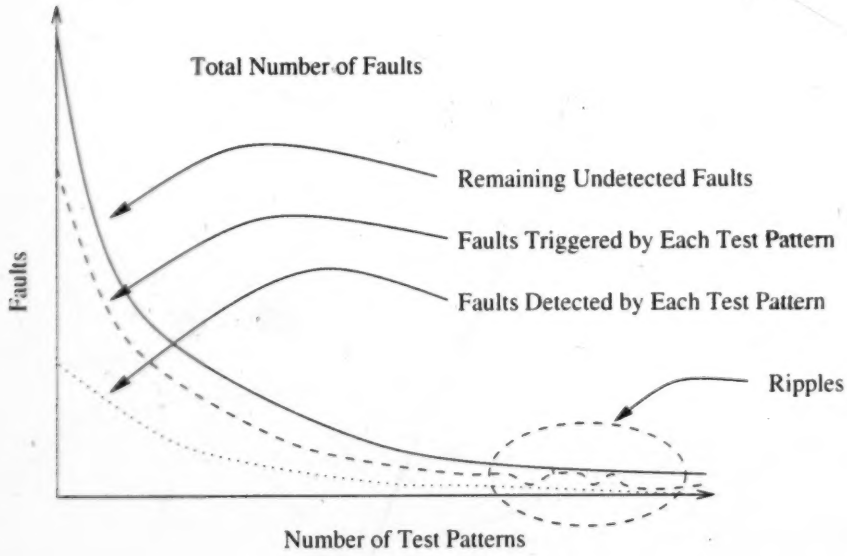


Figure 4.10: Curves for Undetected Faults, Triggered Faults and Detected Faults

test patterns) because of the fault dropping effect. The number of faults being triggered by each pattern tends to decrease as well because most of the easy-to-detect faults tend to be triggered and detected in the early rounds of the simulation. After a while, the curve levels off with a slight ripple remaining due to the fact that some test patterns trigger faults better than others. This is the same for the fault detection curve, which plots the number of faults being detected versus time. Typical curves are shown in Figure 4.10.

4.6.2 Measuring Efficiency - PE Utilization

Pattern-parallel fault simulation is most efficient when there are a lot of faults that are triggered frequently while being very hard to detect. It is wasteful, however, when many test patterns are simulated in parallel and very few of them actually trigger the fault. In addition, even if there are a lot of test patterns triggering the fault, it is wasteful if the very first triggering pattern is capable of detecting the fault, making the rest of the pattern simulations unnecessary. In summary, the simulation of a fault triggering situation is necessary if the pattern that triggers the fault is before or the same as the test pattern that first detects the fault; i.e. if test patterns 15, 25, 43, 67, 80, 190, 253 are all test patterns that trigger a specific fault F, and both pattern 80 and 253 detect F, then the number of *necessary simulations* equals 5, which is the number of test patterns that trigger the fault before and including pattern number 80.

Based on this definition, *PE utilization* is calculated as the number of necessary simulations divided by the number of PEs performing simulations. In the above example, the PE utilization is 5/256, assuming 256 PEs are used for simulation. Here, PE means the actual number of PEs in a C•RAM, or the number of bits evaluated in parallel in the case of a conventional algorithm.

Using the PE utilization figure, we can explain why the PPSFP algorithm is not linearly scalable with the size of the SIMD machine. This is done in Chapter 6.

4.6.3 Another Measurement - Speed-up

Another measurement that we found to be useful in this research is *speed-up*. Speed-up is a measure of how an algorithm performs after some enhancement relative to how it performed previously [16]. Hence, by definition (*Amdahl's Law*),

$$Speedup = \frac{t_s+t_p}{t_s+t_p/N} = \frac{Execution\ time\ before\ improvement}{Execution\ time\ after\ improvement}$$

where t_s is the sequential run time, t_p is the total parallel run time, and N is the number of processors. We will use the PPSFP fault simulator *simf*, described in the next chapter, to give us "before improvement" execution times.

Often an algorithm can be divided into separate modules, and enhancements may only be applicable for some of the modules. In our case, the fault simulation programs

can be divided into two categories: code that can be accelerated by C•RAM and the code that cannot be accelerated. Since C•RAM only accelerates part of the whole process, increasing the size of C•RAM does not imply a linear improvement of the performance. Details for measuring the speed-ups in individual simulators can be found in their corresponding evaluation sections.

A C•RAM emulator is used to emulate the C•RAM operations. The parameters used are listed here:

```
0       non-standard parameters follow
16384   bits of memory
1024    PEs simulated
512     Shift Block Size
15      time per operation ns
120     time per memory row operation ns
0       time per read ns
0       time per write ns
2       operations for free in memory cycle
80      time per broadcast ns
-4      mask to identify data on same cache line

Comments:
Proposed DRAM chip
```

In this research, we are running our fault simulators on a *Sun Ultra-1* workstation, which has a processor that runs at 143MHz. On the other hand, the C•RAM emulator emulates a DRAM implementation, which runs at 120ns per memory row operation (8.3 MHz). This implies a slow-down of roughly 17 times. This setting is used for evaluation in Chapters 6, 7, and 8. In the conclusions, a faster memory technology will be assumed to give a better idea of how much improvement can be obtained.

Chapter 5

simf - a Fast Conventional Pattern-Parallel Fault Simulator

5.1 Overview

The first fault simulator written for this research project implements the Parallel Pattern Single Fault Propagation (PPSFP) algorithm, which runs on a conventional single processor machine[1]. This program is named *simf*. As shown in Figure 5.1, *simf* consists of the main control module and three other modules with specific functions. The ReadISCAS module reads from netlist files in the ISCAS85 format and builds an internal data structure. From this data structure, module GenFault generates a list of potential stuck-at faults, gate-delay faults and stuck-open faults. The list is collapsed to eliminate equivalent faults using the rules described in Chapter 3. The collapsed-fault list and the circuit data structure are then passed to the Simulation module.

[1]Traditional Single Processor machines without C•RAM.

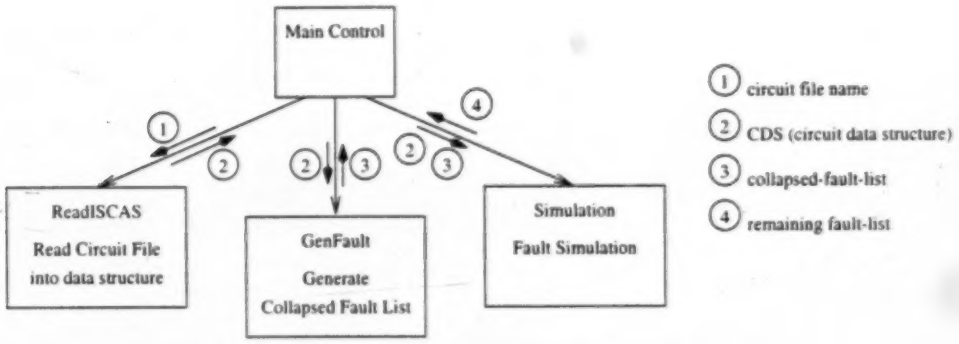


Figure 5.1: Top Level Architectural Design of *simf*

This module reads test patterns from the standard input, performs the fault-free and the fault-inserted simulations, and drops detected faults from the fault list. When either there are no more faults left in the fault list or all the test patterns have been simulated, the simulation is finished. The main control module of *simf* then displays the simulation results.

simf was designed as a reference simulator to represent conventional fault simulators in this thesis. It will be used in the following chapters to evaluate the fault simulators based on the C•RAM architecture. It is thus designed to be as efficient as possible to give a conventional technique a fair assessment. It is also important for the reference simulator to be as accurate as possible. This was made easier because its performance can be compared directly to the public domain fault simulator *sim3*. In fact, *simf* has proven to be so efficient and accurate that it has already been deployed in a separate research project at the University of Alberta to gather simulation data. Evaluation of *simf* is discussed in Section 5.3.

In the following section, design details of three essential components of the fault simulator are discussed. First, the design of the Circuit Data Structure (CDS) is presented in Section 5.2.1. Then the GenFault module is presented (Section 5.2.2). This module generates faults from the CDS and collapses them into a minimum size fault list. The structure of the fault list, which contributes to the acceleration of fault simulation, is also presented in this section. Finally, the Simulation module and techniques used to increase efficiency are presented in Section 5.2.3. The C•RAM versions of the fault-simulators discussed in the following chapters more or less use the same module structure. Changes to these modules, which were made to address the needs of the individual fault-simulators, are discussed in their corresponding chapters.

5.2 Design Details

The Parallel Pattern Single Fault Propagation (PPSFP) algorithm was first proposed as a stuck-at fault simulation algorithm in [31]. In the following year, the algorithm was modified to simulate sequential fault models as well, namely gate-delay faults and stuck-open faults [32]. In this algorithm, each circuit node is assigned an integer. Each bit of the integer represents a different test pattern, with the bit value

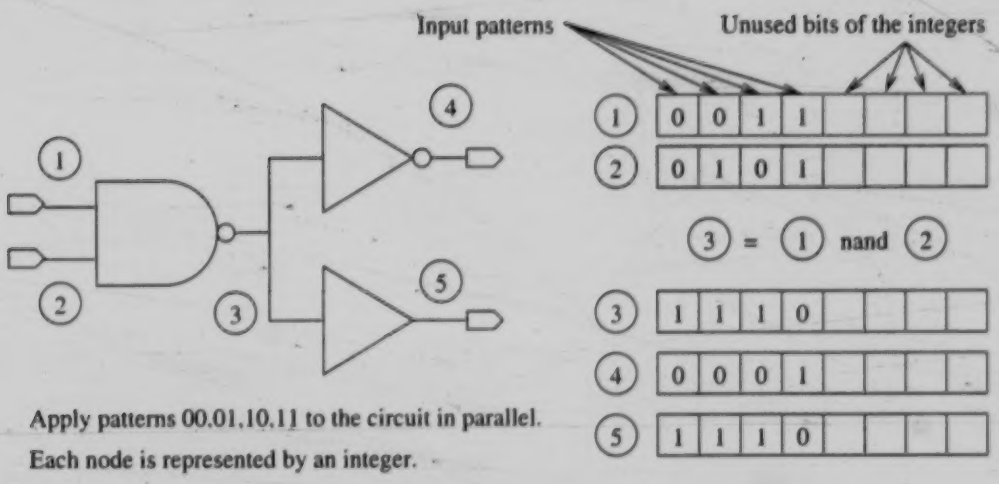


Figure 5.2: Computer Representation of Circuit Nodes in PPSFP

representing the state of the circuit node when the corresponding test pattern is applied. In other words, each bit of the integer represents a different copy of the circuit. Figure 5.2 shows how a circuit and a set of test patterns are mapped into a set of variables.

5.2.1 Circuit Data Structure

In *simf*, a combinational circuit has two types of basic elements, namely logic gates (*gates*) and interconnections (*wires*). The circuit data structure (*circuit*) thus contains an array of gates (*gate_list*) and an array of wires (*wire_list*). In addition, the *input_list* and *output_list* integer arrays are used as indices to point to the primary input and output gates in the *gate_list*. When several interconnections are connected together electrically, the group of wires becomes a circuit *node* that in a good circuit should carry one common voltage signal.

A *gate* is a logical gate of one of the following types: primary input, BUFF, NOT, NAND, AND, NOR, OR, XOR, XNOR. Each *gate* structure contains a field named *gate_type* which denotes the type of the logic gate. In the standard ISCAS85 circuit format, each gate is assigned a gate number (*gate_num*), which is essentially the same as the output *wire number* of the *gate*. Note that this number is different from the *index number* of the *gate*. A *gate* also contains an array of *fan_in* nodes and an array of *fan_out* nodes (see Figure 5.3). The array of *fan_in* nodes is essentially a list of all the

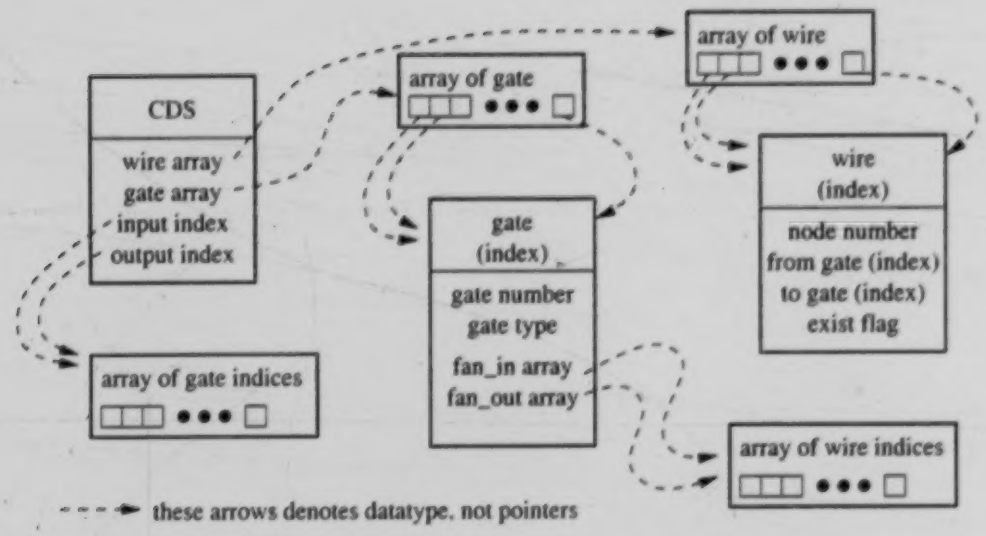


Figure 5.3: Top Level View of the Circuit Data Structure

inputs to the logic gate. The array of *fan_out* nodes is the list of wires that connects the output of a logic gate to other gates. With these two arrays, the connection from the logic gates to the wires is established. If a gate is a primary input, then the input number is stored in a list called *input_num*.

A wire records four pieces of information. The *node_num* variable contains an integer denoting the circuit node this piece of wire is connected to. The *from_gate* variable and the *to_gate* variable contain the indices of the logic gates connected by the wire. With these two variables, the connection from wires to logic gates is established. The index to the *wire* array is the same as the wire number given in the ISCAS85 file. Since the ISCAS85 file format does not require the wire number assignments to be continuous, an *exist* variable is used to mark whether or not the wire really exists in the circuit. This information is necessary for fault generation.

The CDS was designed so that from any gate in the CDS, the wires feeding its input and the gates connecting to its output can be found easily and efficiently. Ease of use and efficiency are especially important for designing algorithms later on in the research that minimize any changes in the CDS that might be required to extend the simulation program.

5.2.2 Fault Generation

After a netlist is translated into CDS, the information is used to generate a fault list. A *fault list* is a list of possible faults associated with a circuit. Three fault models, namely stuck-at, gate-delay and stuck-open, are used to generate these faults. An obvious way to store the faults is to use three separate fault lists. This approach was used in the prototype version of *simf*. However, as discussed in Section 3.5, there are relationships between faults in different models, and it could be advantageous to make use of these relationships. As a result, one single fault list is used to store all three types of faults, which is called the *combined fault list*.

In *simf*, a linked list is used to represent the combined fault list. Each array element, named *Fault*, has six fields: three fields are used to store the information about the fault, while the other three are used to provide the linked list capability. The three fields store the *type* of the fault, which is one of the six possible fault types (SA0, SA1, SF, SR, NO, PO). The *wire_num* variable is used to store the *wire* that the fault is associated with. A *flag* variable is used to tell the status of the fault. The different possible statuses are *fd_UNDETECTED*, *fd_DETECTED*, *fd_IMPLIED*, *fd_EQUIVAL*, and *fd_NONEXIST*. The definitions of these statuses will be given later when recursive fault detection is described. The other three fields used to implement a linked list are: *next*, *implied1* and *implied2*. The *next* variable points to the next element in the linked list. A *Fault* that has the *next* variable pointing to *NULL* signifies the end of the linked list. The *implied1* and *implied2* pointers are used to point to faults that could be implied by the *Fault*. When a fault is detected, all the faults implied by it could also be detected. Marking a fault as detected by implied detection has a much lower cost in CPU time than actually running the simulation to determine whether the fault is detected. Thus by considering the implication relationships between faults, higher efficiency in fault simulation can be achieved.

Figure 5.4 illustrates the fault list of a small circuit. This example circuit is constructed with 2 logic gates and 4 wires. The wire number is not consecutive, as for most of the standard ISCAS85 circuits. The fault list is shown on the right side of the figure. The faults in the list are connected in a linked list fashion (shown with solid arrows). The fault implication relationships of the faults are shown with dashed

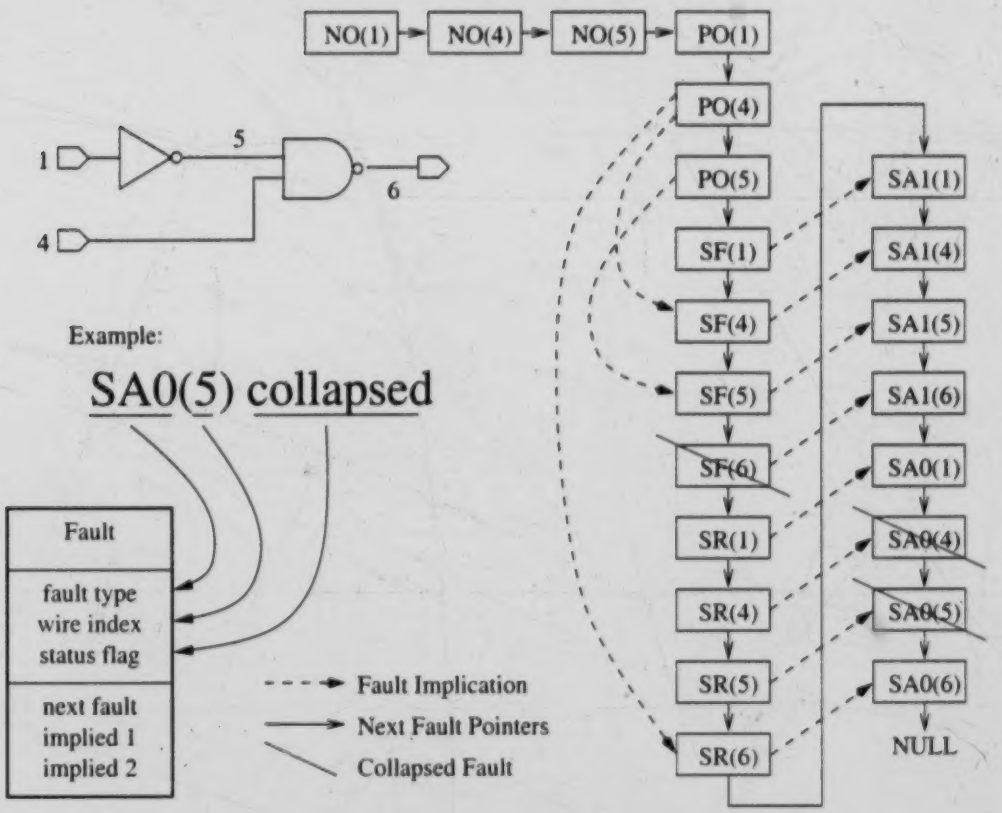


Figure 5.4: A Fault-List Example

arrows. Using the fault collapsing rules described in Chapter 3, three faults are collapsed, namely SF(6), SA0(4) and SA0(5). These faults are marked as *fd_EQUIVAL* in the status flag.

In the *GenFault* module, the first step is to allocate enough memory for the combined fault list. Since each piece of wire in the circuit can have a maximum of 6 faults associated with it, the number of faults before collapsing is less than or equal to six times the *number of wires* in the circuit. In other words, a list whose size in number of records is six times the *number of wires* of *Fault* elements is sufficient for storing the combined fault list. However, since the CDS uses the wire number in the netlist, and the wire numbering in the netlist is not continuous (i.e. wire number 2 may not exist in the circuit), direct indexing into this array of faults will not be possible. Direct indexing is desirable because it provides an efficient way to access

a fault given the *wire number* and the *fault type*. There are two ways to get around this problem: one is to translate the original non-continuous wire numbering into a continuous space, the other is to allocate more memory than is actually needed. The second approach is taken in *simf*, and the amount of *Fault* elements allocated is $6 * (highestwirenumber + 1)^2$.

After allocating memory for the combined fault list, the list elements are initialized. For gate-delay faults and stuck-at faults, all the faults are marked as existing while the remaining faults (including the *Fault* elements that corresponds to non-existing wires and the stuck-open faults) are marked as non-existing (*flag* = *fd_NONEXIST*). After initialization, the algorithm goes through the list of *gates* and collapses the faults using the fault collapsing rules presented in Chapter 3. For the faults being collapsed, their *flag* would be set to *fd_EQUIVAL*. Since stuck-open faults only affect the wires that are inputs to some gates, their *flag* would be set to *fd_UNDETECTED* once they are verified to be gate inputs. In addition, assuming standard CMOS technology, stuck-open faults also affect the internal wires in AND gates and OR gates. These faults are called *internal stuck-open faults*. The internal wires are not part of the wire list, and thus a separate *Fault* elements for these internal faults are not available. Four observations of the properties of stuck-open faults help allocate a *Fault* element for this internal fault without inserting extra internal lines into the CDS: (1) only internal wires of AND gates and OR gates need to be considered (buffers, which are configured as two inverters in series, also have internal wires. However, the stuck-open faults for these wires are collapsed); (2) AND gates and OR gates always have more than one input; (3) there is always at least one stuck-open fault at the input that is collapsed because in AND gates and OR gates, there are at least two transistors connected in series (e.g. the n-transistor open fault (NO) for an AND gate at its second input); and (4) the type of internal fault is the same as this non-existing fault (e.g. the internal fault in an AND gate is also NO). As a result, the collapsed *Fault* element for the AND gates and OR gates is re-used for storing the internal fault. In order to distinguish this fault from the normal stuck-open faults, they are called *f_iNO* (internal NO) and *f_iPO* (internal PO). While collapsing the faults,

[2]In C, since array index starts at 0, allocating an array with X elements will exclude element number X from the array. That is the highest array index would be $X - 1$. Therefore, a +1 term is needed.

fault implication and fault equivalence among fault models are recognized using the *implied1* and *implied2* pointers.

5.2.3 Simulation Algorithm

The basics of simulation in general were described in the last chapter and the basics of the PPSFP algorithm were described at the beginning of this chapter. In this section, more design details on the simulation method used in *simf* will be discussed.

Double Buffered Simulation

The ISCAS85 netlist format has the very important property that the gates in the netlist are recorded in sorted order [3]. A gate list is *sorted* if for all the gates in the list, all the inputs to a gate g must be generated by gates with *gate index* less than that of g. In other words, when the gates in the list are evaluated in the sorted order, it is guaranteed that the states of the inputs to the gate g are all up-to-date when gate g is evaluated. Since *simf* stores the gates in the same order as they were defined in the netlist file, this property is inherited.

This sorted order is very important for doing simulations because a fault-free simulation can simply be carried out by evaluating all the gates in the *gate list* in the defined order. The resulting simplification is illustrated in the following code fragment:

```
for (int j=0; j<number_of_gate ; j++)
    Evaluate_gate(j);   // evaluate the gate

// make an extra copy of the state variables
memcpy (good_value, wire_value, number_of_state_variables);
```

It is also true that when a fault is inserted at gate g, all the gates that are affected by this fault must have a *gate index* greater than that of g because the fault can only affect the outputs of g. Therefore, when a fault is inserted, we can propagate the fault by evaluating only the gates after the point of fault insertion. A lot of simulation effort can be saved using this fault simulation method because all the unnecessary gate evaluations preceding the site of the fault are eliminated. In order to implement this, a double buffered data structure for storing the circuit states was used.

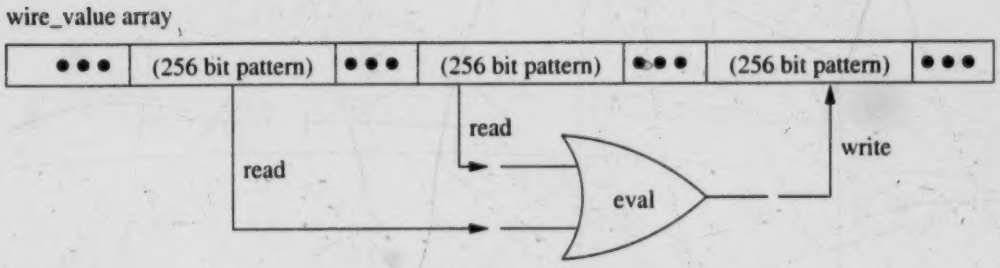


Figure 5.5: Gate Evaluation in *simf*

Two integer arrays are used for storing circuit states, namely *wire_value* and *good_value*. Each array has n elements, where n equals the number of gates in the circuit +1, and each element is an integer array for representing the parallel circuits (*state variable*). When gates are evaluated, circuit states of the gate inputs are read from the *wire_value* array, and the evaluated outputs are also stored in this array (see Figure 5.5). The *good_value* array is always used for storing the fault-free states of the circuit nodes. Keeping the fault-free circuit states around enables us to perform fault simulation downstream from the site of the fault instead of resimulating the whole circuit from the primary inputs. After each round of fault insertion and simulation, the fault-free circuit states can be copied back to *wire_value* for the next round.

Here is how a full simulation takes place. After reading the circuit netlist and generating the collapsed fault list, *simf* is ready to perform fault simulation. The simulation module first reads a set of test patterns from the standard input (*stdin*). The patterns are read directly into the *wire_value* array as the circuit states of the primary inputs. In *simf*, primary inputs are defined as a special kind of gate which has zero input, whereas primary outputs are normal gates with zero fan-out. The program will read in only enough test patterns for one round of simulation; the remaining ones will be read only after the current round is done and there are in fact still undetected faults.

After the next set of patterns is read, *simf* will first do a fault-free simulation. Since the *wire_value* array has already been loaded with test patterns, a fault-free simulation simply evaluates all the gates in the *gate list*, in the order of the *gate index*, skipping over any gates defined as primary inputs. After that, the *wire_value* array is filled with the fault-free states of all the circuit nodes. The content is then

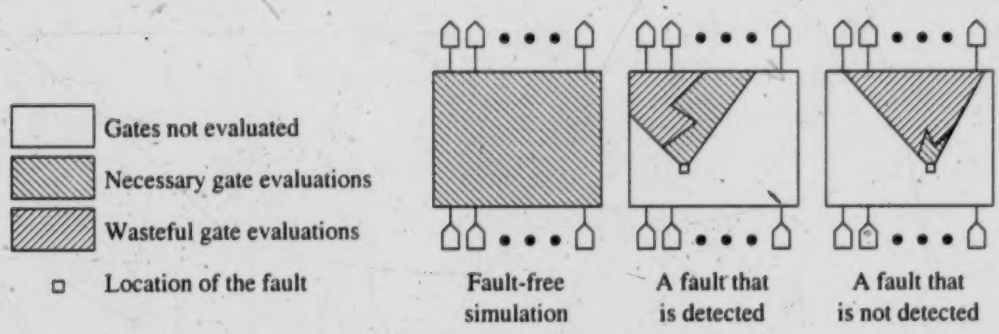


Figure 5.6: Fault Propagation in Circuits

saved to the *good_value* array.

Once the *good_value* array is ready, *simf* starts looking for faults that are triggered by the current set of test patterns. This process is called *fault triggering detection.* The faults that are being triggered will then be inserted into the *wire_value* array for fault simulation.

The *wire_value* array is *clean*, meaning that it stores the fault-free states, before a fault is inserted. After fault insertion, the circuit node where the fault is inserted is the only *dirty* gate in the *wire_value* array. A fault simulation can then be carried out by evaluating all the gates after the dirty gate, i.e. by propagating the fault. If any of the primary output nodes in the *wire_value* array is dirty after this process, then the fault has been propagated to one of the primary outputs, and the fault can be dropped from the fault list. Before going on to insert the next fault, all the dirty nodes in *wire_value* are restored to fault-free states so that the *wire_value* list is clean again. When all the faults in the fault list have been examined and all the fault simulations are done, this round of simulation is finished and another set of test patterns can be loaded if necessary.

The Event Heap

Simulation effort can be reduced by simulating downstream from the point where the fault is inserted. This is because all those gate evaluations before where the fault is located just recompute known fault-free node states. On the other hand, if the effect of the fault stops propagating somewhere before the primary outputs, then all those gate evaluations from the point where the propagation stops up to the primary output

are also wasteful. This waste could be very significant if the fault is near a primary input and its effect stops propagating after a short while afterwards. Figure 5.6 shows these wasteful gate evaluations in a simplified diagram.

In order to eliminate these unnecessary gate evaluations, an *event_list* is used. The *event_list* is simply an array of Boolean variables where each Boolean represents a gate. If the Boolean variable is TRUE, then the gate needs to be evaluated; otherwise, the gate can be ignored. When performing fault simulation, the *event_list* could be initialized so that all the gates except the starting gate are set to FALSE. After evaluating each gate, the state of the gate output is compared with the fault-free state. If they are the same, then the fault effect stops propagates and the fan-outs of this gate don't have to be added to the *event_list*. On the other hand, if the states are different, then we need to add the fan-outs of this gate to the *event_list*, i.e., by setting their corresponding Boolean variable in the *event_list* to TRUE. The following code demonstrates this algorithm:

```
starting_gate = index of the gate with fault inserted;

// set all the elements of event_list to FALSE
memset (event_list, 0, number_of_gate);

for (int j = starting_gate; j < number_of_gate; j++)
{
    if (!event_list[j]) continue;

    Evaluate_Gate (j);

    if (gate output is different from good state)
    {
        if (gate output is a primary output)
        {
            the fault is detected
            stop the simulation;
        }
        else
        {
            for each index of the fan-out gates
                event_list[index of fan-out gates] = TRUE;
        }
    }
}
```

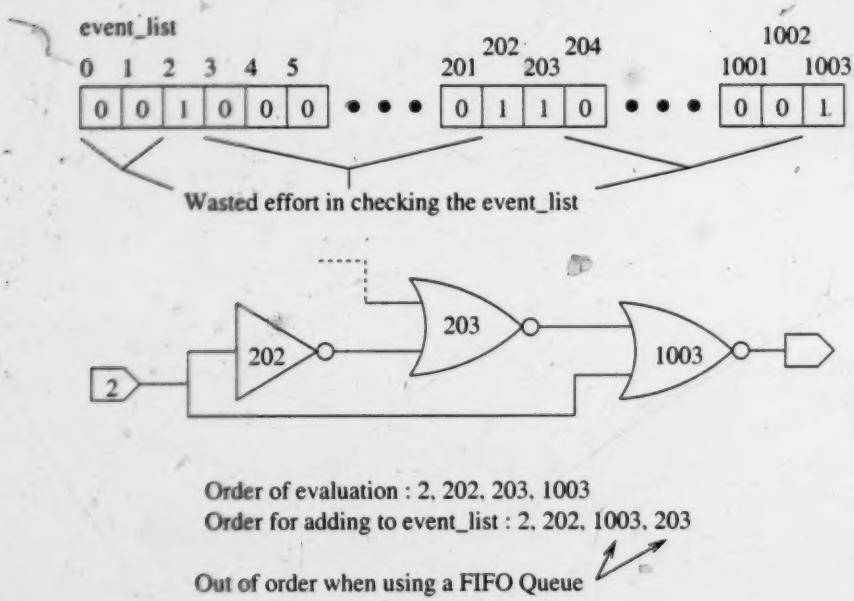

event_list
0 1 2 3 4 5
201 202 203 204
1001 1002 1003
0 0 1 0 0 0
0 1 1 0
0 0 1
Wasted effort in checking the event_list
2
202
203
1003
Order of evaluation : 2, 202, 203, 1003
Order for adding to event_list : 2, 202, 1003, 203
Out of order when using a FIFO Queue

Figure 5.7: Using a Heap for Sorting Simulation Events

```
Restore the fault-free states for the next fault simulation;
```

Using *event_list*, the problems illustrated in Figure 5.6 can be avoided. However, there is still room for improving the algorithm. From the above code, we see that a for loop is used for going through the *gate list*. If there are a lot of gates in the circuit and the fault only affects a small number of gates, then a lot of time will be wasted checking the *event_list* for the gates that are not triggered. A data structure that stores only the *index* of the gates that need to be evaluated is needed. A simple FIFO (First-In-First-Out) queue cannot solve the problem because it doesn't guarantee that the gates will be evaluated in a sorted order.

Figure 5.7 illustrates these problems. A fault is assumed to affect gate 2. After gate 2 is evaluated, gates 202 and 1003 are added to the *event_list*. Next, gate 202 is evaluated. If the fault propagates to gate 203, then gate 203 is also added to the *event_list*. When a simple FIFO queue is used, the gate evaluation order would be 2, 202, 1003, 203. This evaluation order is incorrect as we see that gate 1003 depends on the output of gate 203. To solve this problem, a data structure that sorts the *event_list* efficiently is needed. As a result, a heap data structure was used.

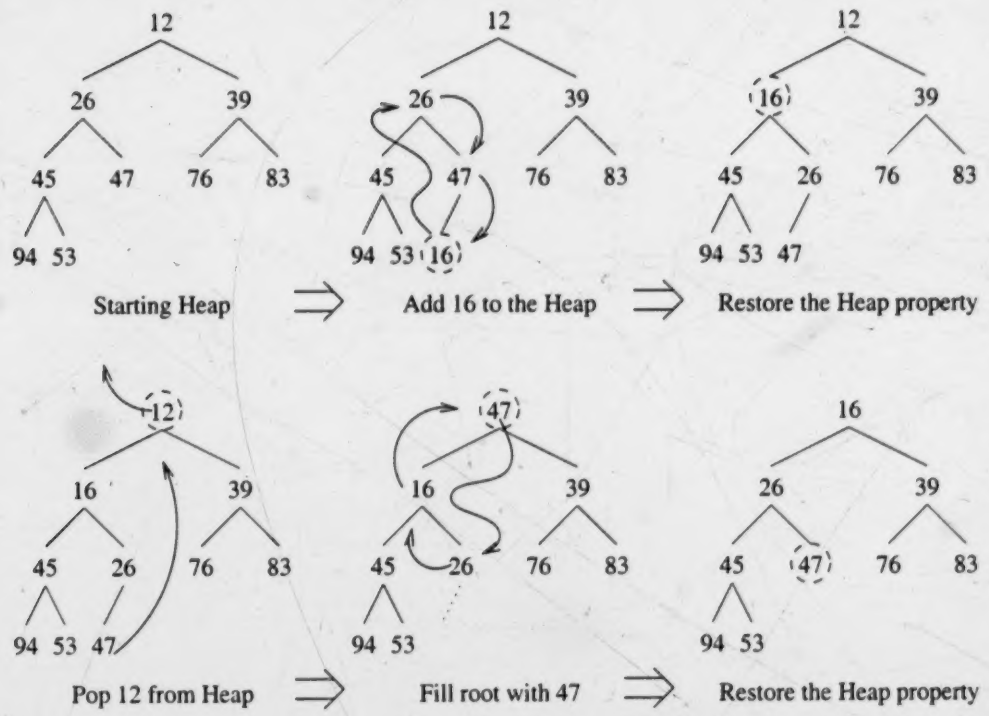


Figure 5.8: Heap - Random Input, Sorted Output

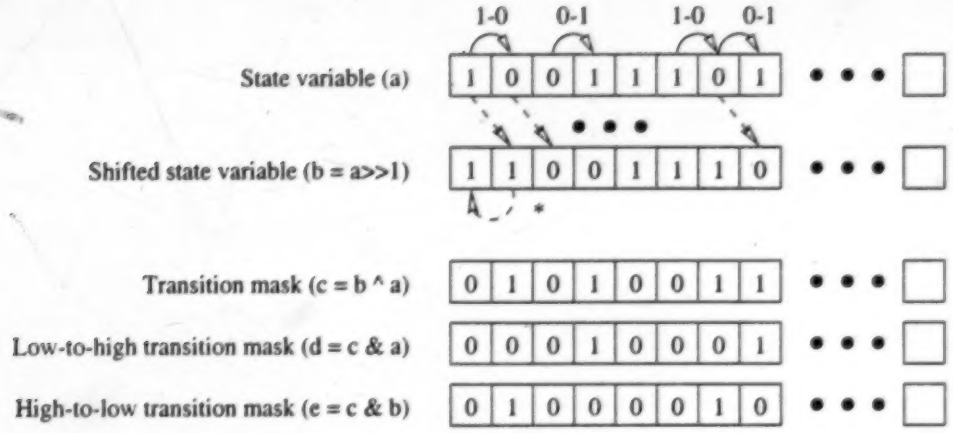
A *heap* is a special kind of binary tree structure, with the added property that the parent index is always smaller than the indices of its children (in the case of a heap that is sorted in ascending order). Another property of a heap is that it is a complete binary tree, meaning that all of its *leaves* (nodes without children) are separated by no more than one level, and all the nodes at the bottom level are left-justified. Figure 5.8 shows the operations of a heap. When an element is added to the heap, it is always added as a new node at the incomplete bottom level on the right-hand-side. The element may have to be added alone to a new bottom-most level. The element's index is then compared with that of its parent. If the parent is greater than the child, then the two nodes swaps the indexes they contain. The node containing the inserted index is then compared with its parent again. This process continues until either the child index is greater than that of the parent or that there's no more parent (i.e. the new element reaches the *root*) When an element is taken out from the heap, it is always popped from the root. The last node of the tree (i.e. the rightmost element at the bottom level) is removed and is placed in the root node position to start the

process of restoring the heap. The element in the root is then compared with its children. If the parent index is smaller than its two children, then nothing needs to be done. Otherwise, the smallest index of the children is swapped with the parent. If a swapping happens, then the new child is compared with its two children again. This process continues until no more swapping can occur. An integer array is used to implement the heap. For more information, please see [11]. Using a heap for sorting the *event_list* on-the-fly, the fault simulation algorithm can be made more efficient.

```
add_to_heap (starting_gate index);

while ((j = pop heap) >= 0)   // get the first element in the heap
{
    Evaluate_Gate (j);

    if (gate output is different from good state)
    {
        if (gate output is a primary output)
        {
            the fault is detected;
            stop the simulation;
        }
        else
        {
            for each index of the fan-out gates
                add_to_heap (index of fan-out gate);
        }
    }
}

Restore the fault-free states for the next fault simulation;
```

Fault Triggering and Fault Insertion

When a set of test patterns is applied to a CUT, the states of the circuit nodes are changed. From these states, we can determine a set of faults that are triggered by these patterns. For example, if a test pattern resets a circuit node to logical 0, then we know that the stuck-at-1 fault associated with this circuit node is triggered (triggering conditions were discussed in Chapter 3). When a fault is triggered, we want to simulate the CUT with the fault effect. The act of adding the fault effect to the circuit is called *fault insertion*.


1-0
0-1
1-0
0-1
State variable (a)
1 0 0 1 1 1 0 1
Shifted state variable (b = a>>1)
1 1 0 0 1 1 1 0
*
Transition mask (c = b ^ a)
0 1 0 1 0 0 1 1
Low-to-high transition mask (d = c & a)
0 0 0 1 0 0 0 1
High-to-low transition mask (e = c & b)
0 1 0 0 0 0 1 0

* For the first pattern, there are two cases:
if there is a preceeding pattern, use that pattern, otherwise, repeat the pattern itself.

Figure 5.9: Detecting Circuit-Node Transitions

To keep things simple, let's consider *fault triggering detection* for the stuck-at fault model only. In PPSFP, the state of each circuit node is stored in one or more integers, where each bit of the integer represents the circuit state corresponding to a different test pattern (*state variable*). Given a stuck-at-1 fault for a circuit node, say, SA1(24), we could simply check every bit of the integers corresponding to circuit node 24. If any of these bits is 0, we know that the fault is triggered. In order to maximize the throughput, we can take advantage of the 32-bit datapath of the processor[3] and use the logical operations provided. For the same example, instead of checking each bit of the integers one by one, we can check the array of integers with a bit-wise AND operation. If the result is not an all-one pattern (0xffffffff), then there must be a 0 in one of these patterns. For stuck-at-0 faults, an OR operation and an all-zero pattern (0x0) could be used.

Fault triggering detection with stuck-at faults is easy, since triggering of stuck-at faults requires only one pattern. However, both gate-delay faults and stuck-open faults requires two consecutive test patterns to trigger the faults, which makes the situation more complicated. Figure 5.9 illustrates the steps needed for detecting transitions. In PPSFP, since each bit of an integer represents a pattern, the consecutive

[3]Assuming a typical present-day 32-bit processor.

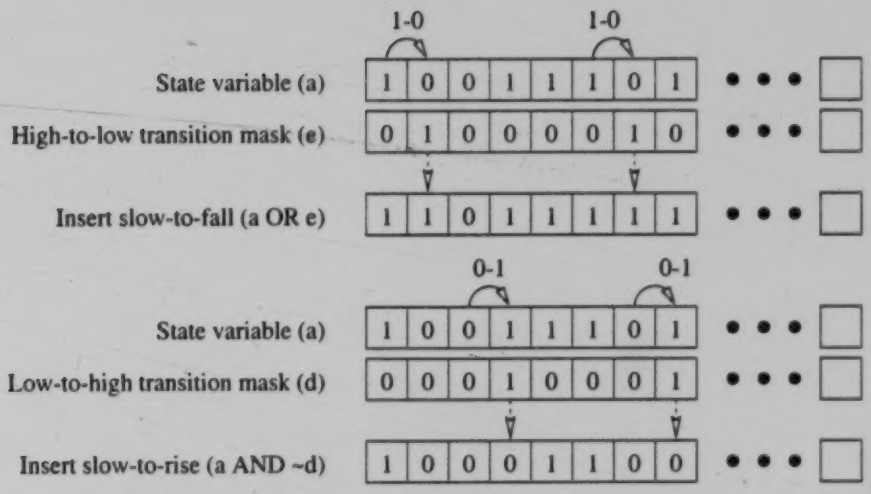

1-0
1-0
State variable (a)
1 0 0 1 1 1 0 1
High-to-low transition mask (e)
0 1 0 0 0 0 1 0
Insert slow-to-fall (a OR e)
1 1 0 1 1 1 1 1
0-1
0-1
State variable (a)
1 0 0 1 1 1 0 1
Low-to-high transition mask (d)
0 0 0 1 0 0 0 1
Insert slow-to-rise (a AND ~d)
1 0 0 0 1 1 0 0

Figure 5.10: Fault Insertion Using Transition Masks

test patterns are located side-by-side in an integer. In order to detect a low-to-high or high-to-low transition on a circuit node (which is required for triggering either gate-delay or stuck-open faults), we need to first shift the *state variable* to the right by 1 bit and then perform an exclusive-OR operation between the shifted and unshifted *state variable*. The result of this operation is an integer where the 1's represent transitions (*transition mask*). If a low-to-high transition is needed, then the *transition mask* is ANDed with the unshifted *state variable* so that all bits representing the high-to-low transitions are masked out (*low-to-high transition mask*).

If the *low-to-high transition mask* (or the *high-to-low transition mask*) does not contain a one, then the fault in question is not triggered. On the other hand, if there is at least one 1 in the mask, then the fault has to be inserted for fault simulation. When performing fault insertion, it is necessary to keep the patterns that do not trigger the fault at their fault-free state while setting those that trigger the fault to the faulty state. For example, a *state variable* may contain both 1's and 0's, and a stuck-at 1 fault needs to be inserted. In this case we want to set all the bits containing 0 to 1 while keeping all those bits that were 1 to remain at 1. In other words, all the bits of the *state variable* are set to 1. In cases of stuck-at 0 faults we can set all the bits to 0. When dealing with gate-delay faults and stuck-open faults, however, it is not that simple because not all the bits containing 1 could trigger a

slow-to-rise faults or PO fault. For these faults, a transition is needed, as mentioned above. Figure 5.10 illustrates our method of fault insertion using transition masks. To perform fault insertion, the *low-to-high transition mask* or the *high-to-low transition mask* is required. Since the 1's in these masks correspond to a transition, the mask can be used to selectively set some bits of the *state variable* to 1 (with an OR operation) or 0 (with an AND operation).

Recursive Fault Detection

After a fault is triggered, inserted, simulated and found to be detected, it needs to be removed (i.e. dropped) from the fault list. The structure of the fault list was described in Section 5.2.2. Usually, this procedure is repeated for each fault in the fault list. However, by exploiting the fault-implication relationships, we can save a lot of the effort in simulation by recursively removing faults from the fault list. We can do so when a fault with implied faults is detected. The following code demonstrates the code for the recursive function.

```
void Mark_detected (Fault *fault)
{
    // fault is detected by implication
    fault->flag |= fd_IMPLIED;

    if (fault->implied1)
      if (fault->implied1->flag == fd_UNDETECTED)
        Mark_detected (fault->implied1);

    if (fault->implied2)
      if (fault->implied2->flag == fd_UNDETECTED)
        Mark_detected (fault->implied2);

    return;
}

...

if (fault is detected)
{
    Mark_detected(fault);
    fault->flag = fd_DETECTED;     // fault is detected
}
```

Here, the *flag* variable in the *Fault* structure plays a very important rule. As mentioned above, the *flag* variable is used to record the status of the fault. Before

Circuit	Number of Roots	Number of Faults	Root/Fault Ratio	Maximum Reduction	Actual Reduction
c1355nr	2405	5238	45.91%	54.09%	48.15%
c1908nr	2988	6401	46.68%	53.34%	24.65%
c2670nr	3523	7446	47.31%	52.69%	0.40%
c3540nr	5725	11833	48.38%	51.62%	28.47%
c5315nr	9072	19261	47.10%	52.90%	35.32%
c6288nr	11220	24962	44.95%	55.05%	60.61%
c7552nr	12177	25579	47.61%	52.39%	2.17%

Table 5.1: Root-to-Fault Ratio for the ISCAS85 Circuits

fault collapsing, all existing faults are marked with *fd_UNDETECTED* whereas the rest of the faults are *fd_NONEXIST*. After fault collapsing, the collapsed faults are marked as *fd_EQUIVAL*, meaning that they are equivalent to some other faults. The fault list contains only faults that are marked as *fd_UNDETECTED*. When a *Fault* is detected, the recursive algorithm first searches for all the faults implied by the *Fault* and marks them as *fd_IMPLIED*, meaning that they are detected by implication. Then the *Fault* itself is marked as *fd_DETECTED*. When all the faults are checked for the current set of test patterns, a procedure will be called to remove from the fault list all the *Faults* marked with either *fd_DETECTED* or *fd_IMPLIED*. As a result, when the next set of test patterns is loaded for simulation, the fault list again contains only all the undetected faults.

Table 5.1 shows the root-to-fault ratio for the ISCAS85 benchmark circuits [3]. A *root* in the fault list is defined as a fault that is not implied by any other fault. The root-to-fault ratio is thus *the number of roots* in a circuit divided by *the number of faults*. In the table, all the ISCAS85 circuits have a root-to-fault ratio near 47%. If a set of test pattern detects all the roots the first time they are triggered, then a maximum simulation reduction of about 53% (100% - 47%) could be achieved. In reality, it is nearly impossible to reach this maximum; however, my experiments have shown that the simulations are still accelerated by an average of 28.5% using this recursive fault detection algorithm.

Circuit	*sim3*	*simf*	Speed-up	# Patterns
c1355nr	2.86	0.64	4.27	1024
c1908nr	5.40	1.50	3.35	3328
c2670nr	76.69	68.02	1.13	256k
c3540nr	11.59	1.69	6.94	1536
c5315nr	6.45	2.05	3.12	1024
c6288nr	36.58	9.45	3.84	256
c7552nr	150.88	71.84	2.10	256k

Table 5.2: Evaluation of Simulation Speed (in seconds)

5.3 Evaluation of simf

simf was designed to be both efficient and accurate. Another fault simulator named *sim3*, which is a public domain fault simulator, was used as a benchmark to evaluate these claims. The source code of *sim3* is not available, so it is nearly impossible to know how it is implemented [34]. As a result, *simf* was designed from scratch, following the basic design concepts shared by all of PPSFP simulators. All the techniques mentioned in the previous section are implemented in *simf*. The seven larger ISCAS85 circuits were used as benchmark circuits for the evaluations.

The UNIX shell command *time* was used to measure the *user time* of the simulators to evaluate the efficiency of *simf*. As shown in Table 5.2, fault simulation times for most of the circuits were sped up with respect to *sim3* by anywhere from 1.13 to 6.94. The speed-up is calculated as (*sim3 time*/*simf time*). These circuits are all simulated to 99% stuck-at fault coverage (except for c2670nr and c7552nr, where 95.89% and 98.75% were obtained, respectively). This shows that *simf* is particularly good at detecting the faults early on in the simulations. One of the reasons for not achieving a high speed-up when there are a lot of test patterns is that the time to read in a test pattern from stdin (standard input) is relatively constant for both simulators. The more patterns that are run, the lower is the contribution of this common constant. In general, *simf* appears to be considerably faster than *sim3*.

In terms of accuracy, the total number of faults after collapsing is an important measure of the accuracy in generating faults. The total number of faults generated by *simf* and *sim3* for each circuit are compared in Table 5.3. Note that for stuck-at

	sim3			*simf*		
Circuit	SA	GD	SO	SA	GD	SO
c1355nr	1566	2076	1596	1566	2076	1660*
c1908nr	1862	2472	2067	1862	2472	2401*
c2670nr	2142	2843	2481	2142	2823*	2837*
c3540nr	3126	4321	4413	3126	4294*	4881*
c5315nr	5248	7303	6754	5248	7259*	7416*
c6288nr	7638	10003	7337	7638	9987*	7337
c7552nr	7041	9525	9024	7041	9514*	10170*

Table 5.3: Fault Collapsing Results for the ISCAS85 Circuits

faults, both *simf* and *sim3* produce exactly the same number of collapsed faults and fault coverage. For gate-delay faults, the number of collapsed faults is different for some circuits. This is because *sim3* collapses gate-delay faults in a slightly different way than *simf*. In *sim3*, gate-delay faults for inverters and buffers are collapsed at the input, whereas they are collapsed at the output for the other logic gates. This results in the situation described in Chapter 3, where some faults that are equivalent cannot be collapsed. Although we don't have the source code of *sim3* to verify this claim, we did modify *simf* in an experiment to collapse faults in this fashion. This modified version of *simf* produces exactly the same amount of gate-delay faults as *sim3*, which provides empirical evidence of how sim3 does its fault collapsing. We should also note that the number of stuck-open faults is also different. This is because *sim3* did not take into account the stuck-open faults at the input of buffers.

Another important measure of accuracy are the simulation results. A small test circuit was created and fault simulated manually in the early phases of the development of *simf*. After getting *simf* to simulate this circuit correctly, it was used for simulating the ISCAS85 circuits and the results were compared with those of *sim3*. As mentioned in the previous paragraph, the total number of delay faults for the two simulators was different. As a result, the fault coverages for delay faults cannot be compared directly. To test the simulation of gate-delay faults, *simf* was again modified to generate the same number of faults as *sim3*. Through all these testing phases, *simf* was found to be accurate in fault simulation.

We can also compare the fault simulator is in terms of supported features. This,

however, was not a high priority in this research. There are significant differences between *sim3* and *simf* in terms of features. For example, *sim3* has three options for feeding the circuit with test patterns: it can either read from the standard input (stdin), read from a file, or generate LFSR patterns by itself. In *simf*, only the first option is supported. The other two options were not implemented because they were unnecessary for the purposes of this research. Another feature is that when simulating a circuit with exclusive-or gates and exclusive-nor gates, which does not have collapsing rules as simple as the other gates, *sim3* will convert these gates into the other logic gates, whereas *simf* will simply ignore them. There were two primary reasons for this decision. First of all, in CMOS technology, exclusive-or gates are not simply a combination of the basic logic gates. In fact, they usually use a special circuit structure called transmission-gates. It was decided that converting the exclusive-or gates to basic gates is simply unrealistic and generates unnecessary faults. In addition, none of the benchmark circuits selected contain either exclusive-or or exclusive-nor gates. Therefore, they are ignored for the purpose of this research. Although these features are lacking from *simf*, this does not diminish its value as a benchmark fault simulator in this research.

Chapter 6

Pattern-Parallel Fault Simulation on C•RAM

6.1 Overview

After developing and validating *simf* to obtain a representative conventional fault simulator, the first fault simulator based on C•RAM was developed. The first simulator, named *simf_pps*, implements a purely pattern-parallel fault simulator. The architecture of this simulator is basically the same as the conventional one, except that instead of using 32-bit integers to store the circuit states and simulating gate evaluations with the logical operations provided by the CPU, the C•RAM processing elements and their local memory, respectively, are used. The simulator uses every PE available in C•RAM to simulate the patterns in parallel, therefore the maximum number of patterns in parallel equals to the number of PEs on the system.

simf_pps reuses all the modules discussed in the previous chapter except the simulation module. A modified simulation module that uses the C•RAM PEs instead of the integer arrays is developed by replacing all of the statements that use integer arrays with the corresponding C•RAM `operate` statements. By doing so, the basic architecture of *simf* is maintained. The following section will discuss the differences between the simulators in more detail.

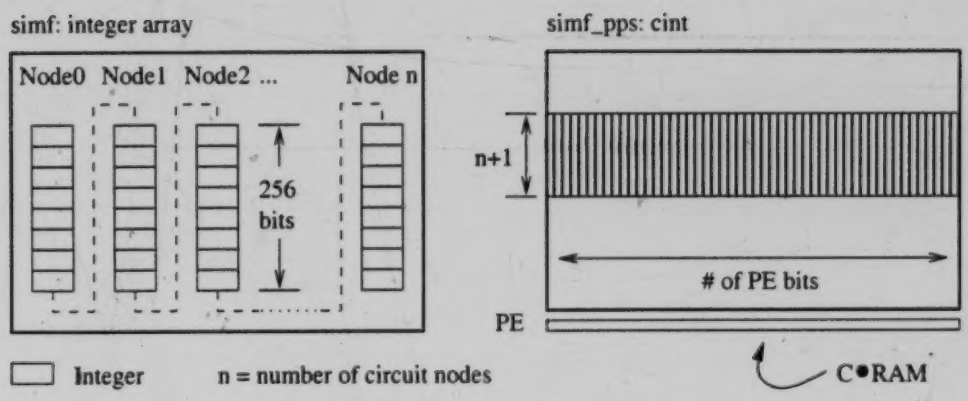


Figure 6.1: Data Structure Used for Storing Circuit Node States

6.2 Design Details

6.2.1 Data Structure

In *simf*, the logical states of circuit nodes are stored in two arrays of integers. The array named *good_value* stores the fault-free value of the circuit nodes when no-fault is inserted; the *wire_value* array is used for performing gate evaluations, fault insertions and fault simulations. When implementing the pattern-parallel fault simulator on C•RAM, these integer arrays are replaced by `cints`, where each `cint` is a C•RAM array of multiple-bit variables with the number of bits equal to the $numberofcircuitnodes + 1$[1] and the number of variables equals the number of PEs. Figure 6.1 shows the difference between these two data structures.

After the fault-free simulation, each fault is checked for fault triggering conditions using the same procedure as *simf*. In *simf*, 256 test patterns are simulated in parallel. However, we often need to simulate more than 256 test patterns, in which case more than one round of simulation is required. Since this simulator implements not only the stuck-at fault model but also the gate-delay and the transistor stuck-open models, the fault-free value of the last pattern in the previous round has to be stored for the fault triggering detection to work correctly. This is because two patterns are required to trigger transition faults. All patterns except the very first one have at least one pattern preceding them, so, if there are n patterns to be simulated, then there are $n-1$ pairs of test patterns that should be considered when checking the fault triggering

[1]The extra node is needed for fault insertion.

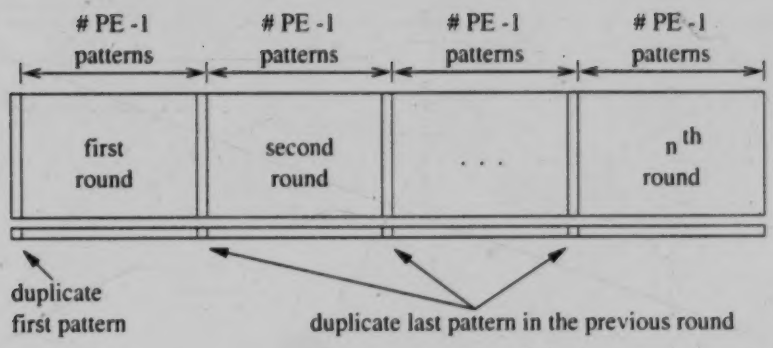


Figure 6.2: Using the First PE to Backup Data

conditions for the transition faults. However, when these patterns are divided into subsets of test patterns for simulating in different rounds, one pair of the test patterns is broken up. When simulating groups of 256 test patterns, the pattern pairs (255[2], 256), (511,512), etc., are broken up. As a result, the fault-free circuit states of the last pattern in the previous round need to be stored aside for correct fault simulations.

In *simf*, tracking the node values for the last pattern in the last simulation round is done by allocating a byte array for backing up the last byte of each *statevariable* in the *good_value* array before loading in the next set of test patterns. During fault triggering condition detection, when the *statevariable* is shifted to the right by one bit, the last bit of the corresponding byte in the back up array is shifted back into the *statevariable* at the left end as the first bit. For *simf_pps*, however, it is not desirable to back up the data in a byte array because this would require m bits to be read from C•RAM during the back up process, and the bits needed to be written back to the C•RAM when checking fault triggering conditions. The extra memory accesses would slow down the whole simulation.

An alternative method, which is implemented in *simf_pps*, is to use the local memory of the first PE to back up the fault-free state of the last pattern in the previous round (See Figure 6.2). Before reading a set of test patterns, the content of the last PE of the *good_value* `cint` is copied to the first PE. Then test patterns can be written into C•RAM starting from PE 1[3]. In practice, only the test pattern in the last PE needs to be duplicated because, after reading all the test patterns, a fault-free

[2]Patterns numbers start with zero.
[3]PE numbers start from zero.

simulation will take place and all the circuit states will be determined, including the first PE.

During the first round, there is no previous round for PE 0 to duplicate. In this case, the first pattern will be written into both PE 0 and PE 1. This is reasonable because the first pattern has no pattern before it, and thus there shouldn't be any signal transition between PE 0 and PE 1. The most efficient way to avoid having transitions there is to make PE 0 and PE 1 store the same circuit states. This is efficient because this way the first PE in the first round does not need to be treated differently.

With the first PE duplicating either the very first pattern or the last pattern of the previous round, the effective number of patterns simulated in parallel is reduced by 1; i.e. if a 1024 PE C•RAM is used, only 1023 test patterns are simulated in parallel in each round.

6.2.2 C•RAM Assembler

All the code in *simf* that uses the integer array data structure is replaced with the corresponding C•RAM assembly code. These code changes are found in three main parts of the simulation module.

Reading Test Patterns

The first section of code requiring changes is the subroutine where test patterns are read from standard input into the data structure. It is assumed that each line of the standard input contains a test pattern. These test patterns are read one bit at a time (a '1' or '0') and are stored in an internal data structure. In *simf*, this data structure is an array of char that is big enough to hold one round of test patterns. For example, if we are simulating 256 patterns in parallel and there are 100 bits per pattern, then the size of the char array is 256×100 bytes. When the primary inputs are evaluated, these test patterns are copied into the good_value array for simulation. In *simf_pps*, the data structure for storing the test patterns is a `cint` whose number of bits equals the number of bits per pattern (i.e. number of primary inputs). Since each pattern is read one at a time, the local memories of the PEs are filled up one PE at a time. The following pseudo code is used to load the test patterns:

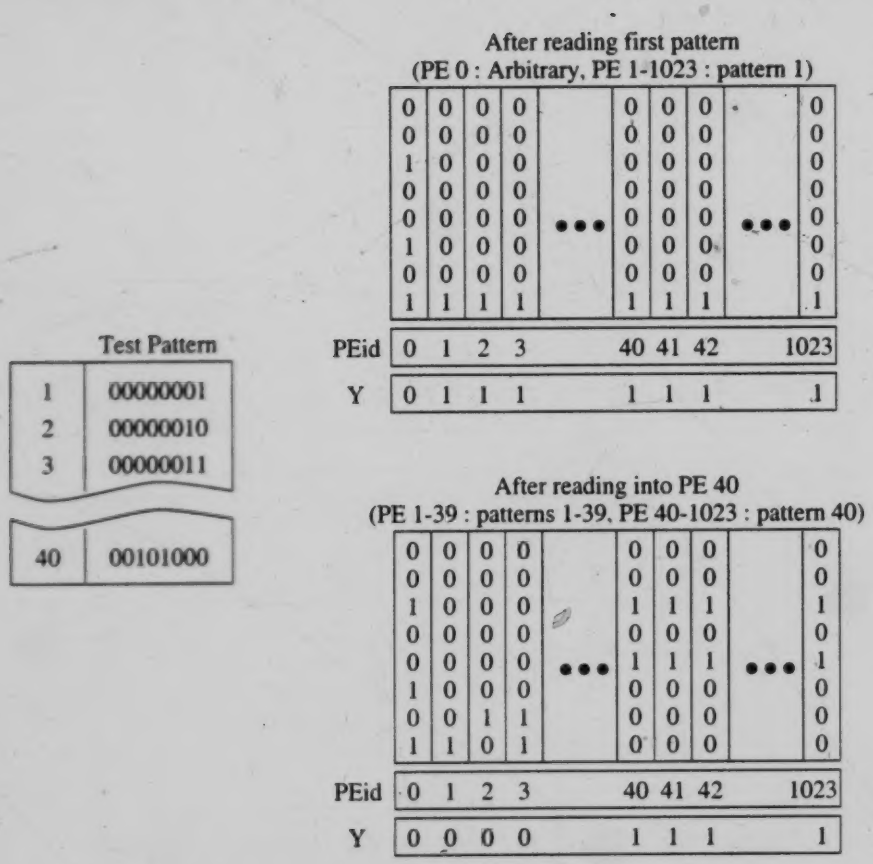


Figure 6.3: Copying Test Patterns into C•RAM

```
copy pattern from PE 1023 to PE 0;

// setup Y register as mask as 01111...1
// * it takes advantage of the fact that the CRAM emulator
//   shifts in 0 for the first PE.
pattern->operate(0, op_one, writey);
pattern->operate(0, op_y, shiftright);

while (read character from standard input)
{
    if (it is a new line character)
    {
        // a pattern is finished reading
        // shift the Y register to the right by one PE
        pattern->operate (0, op_y, shiftright);
    }
    else
```

```
    {
        // put the current character into the X register
        if   (it's 1) pattern->operate (0, op_one, writex);
        else (it's 0) pattern->operate (0, op_zero, writex);

        // store the read character into the appropriate row of
        // local memory, for the PEs where Y register is set to
        // 1 only
        //   opcode 0xe2 = (X & Y) | (M & ~Y)
        pattern->operate (current bit, 0xe2, groupwrite);
    }
}

// copy first pattern to first PE if this is the first time
// pattern is read
if (first time) copy pattern from PE1 to PE0;
```

The Y register here is used as a mask for selecting the PEs to write the test pattern. A value of 1 at one position selects the corresponding PE. Note that all the PEs with a *PEid* greater than that of the desired PE are also selected. As a result, the test pattern will be written to all these PEs. After each test pattern is read in completely, the Y mask is shifted to the right by one bit, so that the previously stored test patterns will not be overwritten. The procedure is illustrated in Figure 6.3.

An alternative way of pattern generation is to use *PEid* as random seed to generate test patterns within C•RAM. This method should be able to save the time needed to read in test patterns one bit at a time. However, the quality of such test pattern generator is unknown. Whether this method can produce test patterns generated by other test pattern generators (such as LFSR) is also an unanswered question. It is an entirely new research topic to generate test patterns using *PEid* as random seed. Since this method is not compatible with *simf*, we did not implement it in this research.

Gate Evaluation

Another portion of the program where C•RAM operations replace traditional ones are the gate evaluation routines. In both *simf* and *simf_pps*, each basic logic gate has a gate evaluation function associated with it. When the gates in the gate list need to be evaluated, their corresponding gate evaluation functions are called. Each such

function basically loads the states of each of the gate's inputs, evaluates the result based on the logical operation the gate represents, and then writes it back to the gate's output node. In the conventional *simf* version, this step has to be repeated for each integer in the integer array that stores the *state value* of the circuit nodes. A simple comparison of the evaluation of NAND gates is presented below:

```
// NAND gate evaluation in conventional simulation
void g_nand (int  j)
{
    // n = number of integer per integer array
    for (int k=0; k<n ; k++)
    {
        // set output state to be the same as the first input's state
        WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
            WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

        // apply AND operation
// to the output state and the rest of the inputs
        for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
            WIRE_VALUE(g_cut->gate_list[j].out_node,k) &=
                WIRE_VALUE(g_cut->gate_list[j].in_node[l],k);

        // negate output so that it becomes a NAND function
        WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
            ~WIRE_VALUE(g_cut->gate_list[j].out_node,k);
    }
}
```

```
// NAND gate evaluation using C*RAM
void g_nand (int  j)
{
    // set X register to be the same as the first input's state
    wire_value->operate
        (g_cut->gate_list[j].in_node[0], op_m, writex);

    // apply AND operation
    // to the X register and the rest of the inputs
    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
      wire_value->operate
          (g_cut->gate_list[j].in_node[l], op_xandm, writex);

    // store X bar at the output of the gate
    wire_value->operate
        (g_cut->gate_list[j].out_node, op_xbar, groupwrite);
```

}

Fault Triggering Detection and Insertion

C•RAM **operate** statements are also used in fault triggering, detection and insertion. As mentioned in the previous chapter, fault triggering detection is done by checking the state variable at the location of the fault in question. In *simf*, a bit mask has to be applied when checking any one of the six types of faults (SA0, SA1, SR, SF, NO, PO) because we only want to detect faults where the test pattern is valid. For example, if only 40 test patterns are in the simulation, we want to ignore the 216 bits of the state variables that are not simulating valid test patterns. In *simf_pps*, this is still true for the delay faults and stuck-open faults because we need to detect signal transitions in order to detect the triggering conditions of these faults, which requires shifting of the state variables to the right by one bit. Since the left-most bit is always shifted in as a 0 (one of the C•RAM configuration options chosen for this project), the first PE could erroneously produce an incorrect signal. Therefore we want to mask out any signal from the first PE. For stuck-at faults, however, no masking is needed. This is because all the PEs are initialized using valid test patterns, although there are repeated ones. To compare the difference between the conventional simulator and the C•RAM version, the code fragments for detecting stuck-at-1 fault triggering for both programs are listed.

```
    // stuck-at-1 fault triggering detection
    // in conventional simulation
    case f_SA1:
    {
        int  flag=0;

        // fault triggering
        // if any of the bits in the good_value state variable is 0
        //    then the fault is triggered.
        // or_mask is a 256 bit mask
// where 0 represents a valid test pattern
        //    and 1 represents invalid ones.

        for (k=0; k<NUM_PV/INT_SIZE ; k++)
            flag |= ~(GOOD_VALUE(fault_node, k) | or_mask[k]);
```

Circuit	*simf*	*simf_pps*	Speed-up	# Patterns
c1355nr	0.641s	0.190s	3.37	1024
c1908nr	1.497s	0.259s	5.77	3328
c2670nr	68.021s	6.675s	10.19	262144
c3540nr	1.689s	0.358s	4.71	1536
c5315nr	2.050s	0.443s	4.63	1024
c6288nr	9.447s	1.730s	5.46	256
c7552nr	76.836s	9.170s	8.38	262144

Table 6.1: Comparing *simf* and *simf_pps* in Terms of Speed-Up

```
    if (flag)
    {
        the fault is triggered.  prepare for simulation
    }
}


// stuck-at-1 fault triggering detection using C*RAM
case f_SA1:
{
    // fault triggering
    // if any of the bits of the state variable is 0, flag = 0

    boolean flag = true;
    good_value->operate (fault_node, op_m, busaccess, flag);

    // setup wire value
    if (!flag)
    {
        the fault is triggered.  prepare for simulation
    }
}
```

6.3 Evaluation

After verifying that the C•RAM version of the pattern-parallel fault simulator produced correct results, the program was ready for performance evaluation. This program was compared to *simf* in two respects: speed and PE utilization. The time measured by the existing C•RAM emulator measures the amount of time it takes to

Circuit	Total Triggering	256 PE		1024 PE		Util. Ratio
		Needed/Used	Util.	Needed/Used	Util.	
c1355nr	63k	39k/797k	4.91%	32k/2108k	1.54%	3.2
c1908nr	349k	323k/2421k	13.36%	273k/4523k	6.03%	2.2
c2670nr	44m	44m/194m	22.51%	44m/197m	22.17%	1.0
c3540nr	390k	318k/3695k	8.62%	257k/7035k	3.65%	2.4
c5315nr	575k	410k/4463k	9.18%	219k/8544k	2.56%	3.6
c6288nr	60k	23k/2514k	0.93%	23k/10058k	0.23%	4.0
c7552nr	33m	33m/151m	21.73%	33m/170m	19.28%	1.1

Table 6.2: Comparison of PE Utilization for *simf* and *simf_pps*

run the C•RAM operations plus the time spent in the performing sequential operations such as organizing the fault list. The results for the ISCAS-85 benchmarks are presented in Table 6.1, where the C•RAM had 1024 PEs.

Another measurement is the notion of *PE utilization*, which was introduced in Chapter 4. PE utilization is measured by implementing an *instrumented version* of *simf* that has more features than the normal one (and is hence slower). One of these features is used to calculate the PE utilization. For example, by setting this version to simulate 256 test patterns in parallel, the PE utilization figure for 256 PEs can be measured. A higher utilization implies less processing power is wasted. Table 6.2 shows the PE utilization for running 256 patterns and 1024 patterns in parallel, which represents *simf* and *simf_pps*, respectively.

When measuring PE utilization, we must record the number of useful simulations versus the total number of simulations done. In theory, the number of useful simulations should be independent of the number of patterns simulated in parallel. This is because the area under the curve of faults being triggered is the same no matter how many test patterns are run in parallel (see Section 4.6 for more detail). However, this is not the whole truth in our case because fault implication is implemented. With fault implication, simulations can be eliminated by realizing that some faults (such as stuck-at-1 faults) can be dropped from the fault list when another fault is being detected (such as slow-to-fall faults). In Table 6.2, we can see that the *needed* columns under 256 PE and 1024 PE are indeed mostly different.

Note that the PE utilization ratio is generally very low (as low as 0.23%). When

1024 test patterns are used, the worst case in PE utilization is that the first or second PE detected the fault. In that case, only 1 out of the 256 PEs is useful, and the PE utilization ratio is about 0.4%. When 1024 PEs are used, the worst case is about 0.1%. From Table 6.2, we can see that the results for the c6288nr circuit is very close to this worst case situation. The PE utilization ratios of our benchmark circuits are expected to be low because the circuits are relatively easy to detect (i.e. they take less than 5k test patterns to reach 99% fault coverage in five of our benchmark circuits). A PE utilization ratio that is greater than 20% is thus a very good ratio.

We should also note that the utilization ratio does not necessarily equal the speed ratio between the two simulations because of three reasons. First, there is a difference between the instruction speed of the CPU and the vector operation speed of C•RAM. Second, when 1024 PEs are used for simulation, only 1023 patterns are actually simulated in parallel. Lastly, a point which is not obvious is that a fault simulation session with 1024 patterns in parallel could potentially run faster than one that has only 256 test patterns. This is an effect of event-driven simulation. Let's use an example to illustrate this point: When 256 test patterns are simulated in parallel, fault A is detected at primary output 20 with pattern 123; when 1024 patterns are used, fault A could be detected at primary output 1 with pattern 923. Assuming primary output 1 is near the beginning of the gate list, whereas primary output 20 is near the end, a lot of gate evaluations can be avoided entirely by stopping the fault simulation early.

6.4 Discussion

In our implementation, two arrays of `cbooleans` are used for storing good circuit states and faulty circuit states. Using two arrays can save a lot of time in simulation because when faults are inserted, we can simulate from the site of the fault and do not have to simulate all the way from the primary inputs. Also, we can stop the simulation as soon as the fault effect stops propagating. However, using two arrays also requires more memory for storing the circuit node values. For example, when simulating a circuit with 10k circuit nodes and 256 patterns in parallel, the amount of memory required will be $2 * 10k * 256/8 = 640kbytes$. This is not an issue in *simf*

because the workstation used is configured with hundreds of megabytes of virtual memory. For *simf_pps*, however, since no virtual memory is implemented for the C•RAM, the amount of C•RAM memory available for programming is limited to the total amount of local memory the C•RAM chip has. Using the same example, a C•RAM chip that has at least 20k bits of local memory per PE is required. However, the biggest implementation of C•RAM at the time of this writing has only 16k bits of local memory per PE. Therefore, the biggest circuit *simf_pps* could support should have less than 8k of circuit nodes. Currently, integrated circuits with more than 8k of nodes are not uncommon. This limitation renders *simf_pps* impractical for realistically large industrial circuits, at least in this simplest version.

One way to make this simulator useful for larger circuits is to group multiple PEs together to increase the local memory size. For example, when two PEs are grouped together, the effective memory size for the PE group is twice that of a single PE. If each PE has 16k bits of local memory, then a group of two PEs will have 32k bits of local memory, which is enough for the circuit described in the previous paragraph. The cost of this method is that half of the PEs are effectively disabled, thus reducing the parallelism. Also, an extra cycle has to be spent when data from a PE's local memory has to be transferred to the next PE in the same group. The larger the PE group, the higher the penalty. Nevertheless, this could be a viable method for simulating larger circuits.

Chapter 7

Fault-Parallel Fault Simulation on C•RAM

7.1 Overview

The second algorithm implemented on C•RAM is Fault-Parallel Fault Simulation, which we call *simf_fps*. In this simulator, C•RAM is still responsible for performing gate evaluations; however, the PEs are no longer simulating the same fault simultaneously. Rather, all the PEs are simulating the same test pattern at the same time for different faults. Since fault-parallel fault simulation is quite different from pattern-parallel fault simulation, a new algorithm needed to be designed. The key differences between this simulator and the previous ones are (1) the use of fault-families in the fault list organization; (2) the use of the conventional CPU for fault-free simulation; and (3) multiple fault insertion. A *fault-family* is a group of faults that can cause a circuit node to appear with the same erroneous value.

Figure 7.1 shows the data flow diagram of *simf_fps*. In this diagram, both the control flow and the major data structures are shown. After reading a test pattern from the primary input, a simulation round begins. A fault-free simulation of the test pattern is carried out by the host CPU, and the data is stored in a data structure, which is labeled *Good Sim Result* in the figure. Note that in *simf_pps*, the fault-free simulation was done in C•RAM. Since the result does not reside in the C•RAM, a module called *C•RAM_mapper* is used to load the node values from the host's memory into C•RAM whenever the data is needed. After the fault-free simulation terminates, the heap is prepared for fault simulation. At this stage, each fault-family is checked against the fault-free circuit's node states to determine whether the fault-family needs

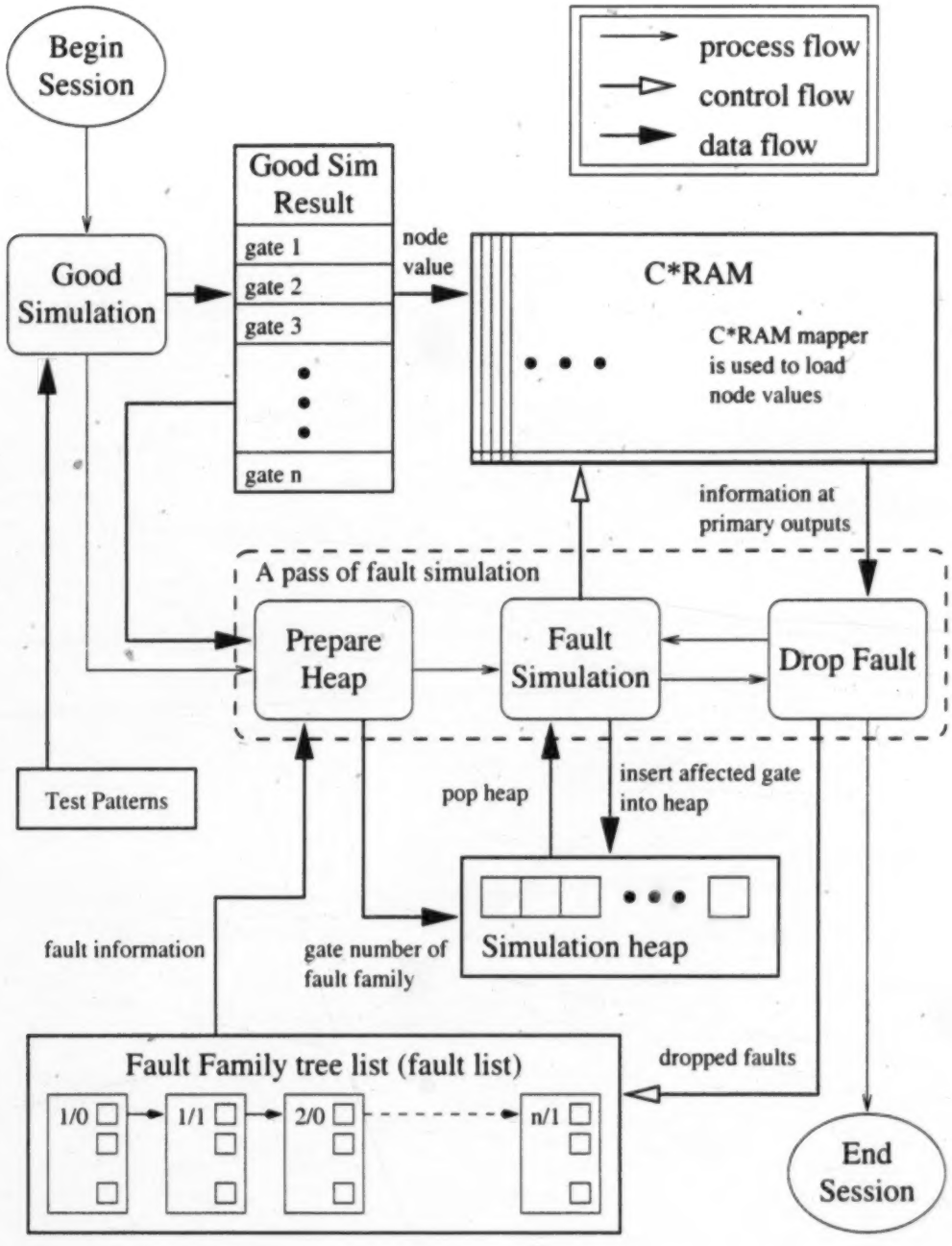


Figure 7.1: Dataflow Diagram of *simf_fps*

Circuit	CPU (ms)	C•RAM(ms)	Ratio
c1355nr	0.183	0.532	2.9
c1908nr	0.346	1.039	3.0
c2670nr	0.482	1.362	2.8
c3540nr	0.627	1.911	3.0
c5315nr	0.914	2.916	3.2
c6288nr	0.878	2.448	2.8
c7552nr	1.320	4.067	3.1

Table 7.1: Speed Difference in Fault-Free Simulation

to be simulated. Then, fault simulation is carried out by the C•RAM. The result of the simulation is then checked to see if any fault-family is detected. If so, the triggered faults in the family are removed from the fault list. If the fault-family contains no more undetected faults, then the family itself is removed from the *fault-family tree list*. When simulating large circuits, it is likely that there will not be enough PEs for fault simulation in one C•RAM load. If this is the case, then fault-families that weren't simulated in the previous round will be added to the simulation heap for simulation in the next round. The current round of simulation is completed when all the triggered fault-families are simulated. Note the three steps surround by a dashed box. This box represents a simulation pass. When there are more fault families needed to be simulated than the amount of available PEs, it is necessary to run multiple passes of fault simulations in one round. In the next section, the design of the fault-parallel algorithm will be discussed in more detail.

7.2 Design Details

7.2.1 Simulation with C•RAM

In *simf_fps*, the fault-free simulation uses the host CPU to perform gate evaluations and stores the result in main memory; however, in *simf_pps*, the C•RAM PEs and their local memory are used. From Table 7.1, we can see that the host CPU runs about 3 times faster than C•RAM when there's only one single test pattern to be fault simulated. Since the fault-free simulation data is accessed very often by the host CPU, it is advantageous to store it in main memory.

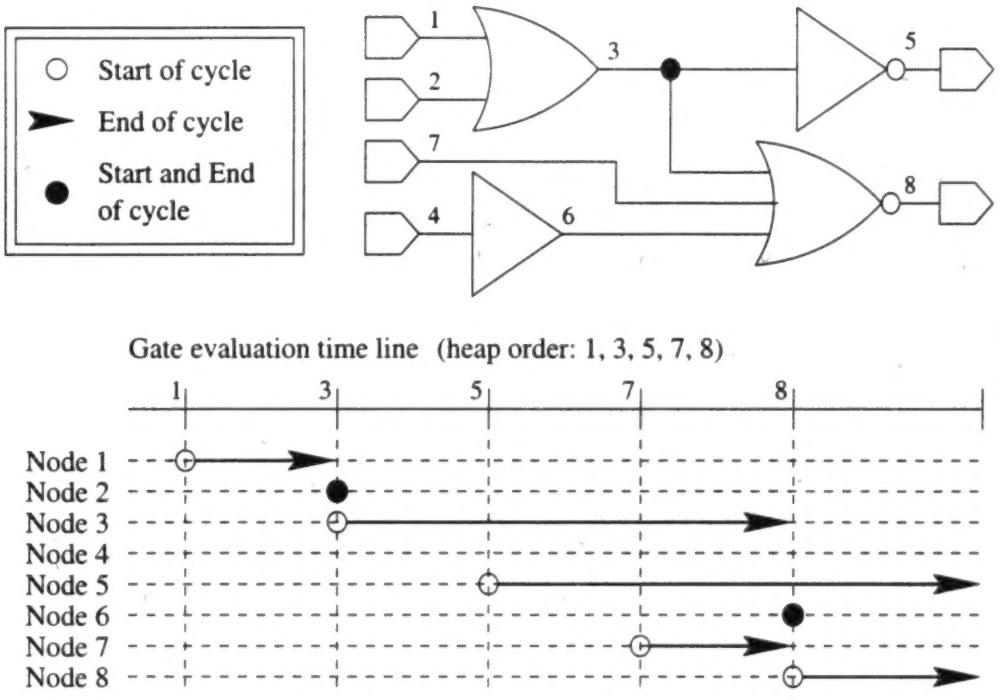


Figure 7.2: Illustration of Circuit Node Life-Cycles

Fault simulation is carried out in the C•RAM. In our simulator, a dynamic memory allocation scheme was implemented; i.e. a circuit node occupies C•RAM memory only for the time when it is needed for fault simulation. It is observed that during a fault simulation, each circuit node passes through a life cycle. A circuit node value can come into existence in two ways. First, when a gate is first evaluated, the gate output value needs to be stored somewhere. This is when an output node of a gate becomes *active* (starts occupying memory). Second, when a node's value is needed for fault simulation and it hasn't been loaded into C•RAM yet, then C•RAM space has to be allocated for it and the node becomes active. The circuit node occupies C•RAM memory until it is not needed anymore. Nodes that are not primary outputs are active as long as they are inputs to at least one gate that has not been evaluated yet. A *reference number* is associated with each circuit node. The reference number is the maximum gate number among its fan-out nodes. When

Circuit	a	b	c	b/a	c/a	Reduction
c1355nr	587	104	33	17.7%	5.6%	94.4%
c1908nr	1194	312	56	26.1%	4.7%	95.3%
c2670nr	1670	310	68	18.6%	4.1%	95.9%
c3540nr	2476	509	181	20.6%	7.3%	92.7%
c5315nr	2431	286	140	11.8%	5.8%	94.2%
c6288nr	911	182	46	20.0%	5.0%	95.0%
c7552nr	3604	623	210	17.3%	5.8%	94.2%

a) Without dynamic memory allocation (number of circuit nodes).
b) Using dynamic memory allocation without sorting.
c) Using dynamic memory allocation with minimum-active-nodes sorting.

Table 7.2: Memory Requirement With or Without Dynamic Memory Allocation

the *current gate evaluation number* exceeds this reference number, the allocated C•RAM memory can be freed and the circuit node becomes *deactivated.*

Figure 7.2 illustrates the concept of a circuit node life cycle. The circuit is simulated with the heap containing nodes 1, 3, 5, 7, and 8. When each of these gates is evaluated, the inputs of the gates will need to be loaded. To satisfy this requirement, it can be seen in the timing diagram that nodes 2 and 6 are activated just before gates 3 and 8 are evaluated. These two nodes are immediately freed, or deactivated, as soon as the gate evaluations are done, because they are not needed anymore. We should note that node 4 is never activated in this setup although it is an input to node 6. This is because we have already determined the value of node 6 in fault-free simulation. In addition, node 5 is not freed once activated because it is one of the primary outputs.

The dynamic memory allocation scheme was implemented so that fault-free node values are loaded into C•RAM only when they are needed. It has the side advantage of reducing the amount of C•RAM memory that is needed for fault simulation, thus allowing the simulation of larger circuits.

Table 7.2 shows the memory requirement for three cases: (a) not using dynamic memory allocation at all; (b) using dynamic memory allocation with unsorted gate numbers; (c) using dynamic memory allocation with gate numbers sorted by minimizing the number of nodes that are active at any time (minimum active nodes).

Memory reduction of up to 96% can be achieved. Sorting gate numbers in *minimum-active-nodes* is an NP-complete problem[1]. The data shown in the table was obtained using heuristics[2], which do not guarantee that an optimal solution will be found. Finding the best heuristic, however, is outside of the scope of this research.

A module named *cram_mapper* was created for maintaining links between the fault-free node values and the C•RAM. It provides six routines. An *Init_CRAM_Mapper* function must be called once in the program before using this module. This routine initializes the necessary data structures. The *cram_mapper* module defines a *CRAMNode* structure which allows the creation of a `cboolean` linked-list. A *node_mapper* array is defined for associating circuit nodes with CRAMNodes. Four functions are provided for accessing the CRAMNodes, namely *Obtain_Node*, *Obtain_Node_w_Loading*, *Obtain_Node_f_mapper* and *Use_Node*. The following code demonstrates inverter evaluation using two of these functions.

```
inline void g_not  (Circuit *cut, int  j)
{
    cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

    cboolean *in_node  = Use_Node (cut->gate_list[j].in_node[0]);
    in_node->operate (0, op_m, writex);

    out_node->operate (0, op_xbar, groupwrite);
}
```

Here, *Obtain_Node* is called to allocate a node from C•RAM. When Obtain_Node is called, the cram_mapper module first checks node_mapper. If a node is already associated with the gate number requested, then this node is returned. Otherwise, C•RAM memory for a new CRAMNode is allocated[3], and node_mapper[gate number] is set to point to this new CRAMNode. The pointer of the new `cboolean` variable is also returned. Once the output node is allocated, *Use_Node* is called to 'use' the input node value. Again, the node_mapper array is checked. If there isn't already a CRAMNode for this node, then a new CRAMNode is created and the `cboolean`

[1] Arbitrary instances of an NP-complete problem cannot be solved deterministically in polynomial time.

[2] Nodes were reordered by analyzing the fan-in and fan-out values.

[3] The present program will fail if there is not enough C•RAM memory.

is initialized with the fault-free circuit node value. This is done by using either the op_one or op_zero C•RAM operation, which sets a `cboolean` to either all 1's or all 0's, respectively. Before returning the `cboolean` for gate evaluation, the reference number of the node is checked. If the current gate evaluation number equals the reference number, then the node is put on the *avail* linked-list so that it is available for future use.

There are two other Obtain_Node functions. The *Obtain_Node_w_Loading* function not only allocates memory for a node, but also initializes it with the fault-free node value. This function is useful during fault simulation, where the fault-free circuit node is needed as input to some other gates. The *Obtain_Node_f_mapper* function simply returns a NULL value when there isn't a CRAMNode associated with the node, rather than allocating memory for it. It is useful when the primary output values are checked for detected nodes, where primary output values that are not affected by faults can be ignored. Finally, there's a *Free_All_Node* function which moves all the CRAMNodes to the avail linked-list for the next simulation round to begin.

With the *cram_mapper* module, fault-parallel fault simulation is carried out in C•RAM. The fault simulation process is described in detail in the following subsection.

7.2.2 Fault Simulation

Fault-Family

So far, faults have been associated with individual wires. Each wire can be affected by up to six faults (SA0, SA1, SR, SF, NO, PO) and each fault is inserted and detected as a separate entity. When multiple wires are connected together, they form a circuit node, which is either the output of a logic gate or a primary input. Thus we have a *node-wire-fault* hierarchy.

Treating faults as individual entities was adequate in the previous simulators where only one fault is inserted in each pass of fault simulation. When there's only one fault, the *primary effect*[4] of the fault can be calculated and inserted as erroneous node values before the simulation takes place (simulating downstream from the fault). If there are multiple faults to be simulated, i.e. in fault-parallel fault simulation,

[4]The primary effect of a fault is the behaviour of the first node being affected by the fault.

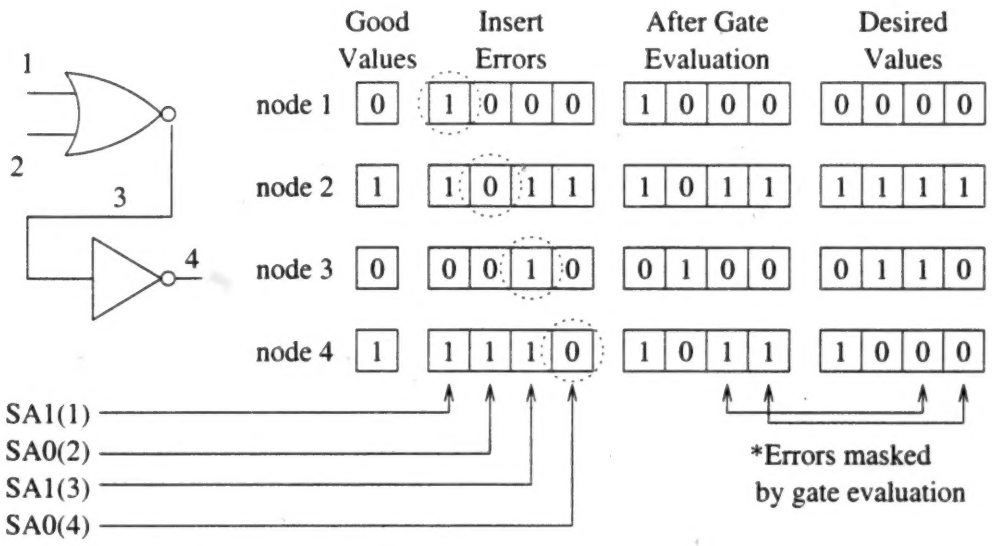


Figure 7.3: Gate Evaluation with Multiple Fault-Insertion

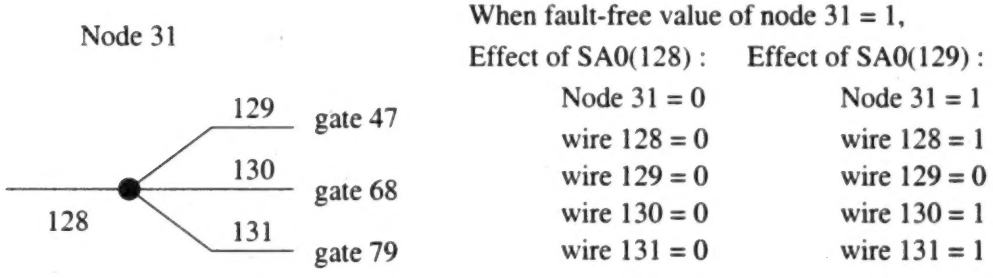


Figure 7.4: Problem with Fan-out Nodes

however, this simple fault insertion method is not adequate. Let's take the circuit in Figure 7.3 as an example. Four faults are inserted to their corresponding wires for fault simulation. Ideally, three faults will be detected (SA0(2), SA1(3), SA0(4)). However, the evaluation of the OR gate erases the fault effect inserted for wire number 3, and the evaluation of the inverter erases that of wire number 4. As a result, only one fault is detected (SA0(2)). In order to avoid this, fault effects must be inserted *after* the gate has been evaluated, this requirement calls for a way to address faults with respect to the gate number.

Our simulator is *node-based*, meaning that the state of a group of wires connected together as a node is represented by a single Boolean variable. This method is far more efficient than a *wire-based* simulator because there is less memory transfer. However,

using a node-based simulator increases the difficulty in treating faults as separate entities. Figure 7.4 illustrates this problem. Node 31 is shown as being composed of the four wires 128, 129, 130, and 131. When there is only one fault (SA0(129)) to simulate, a temporary node can be allocated to store the fault, keeping the original node value unchanged so that gate 68 and gate 79 can be evaluated correctly. With multiple fault insertion, this method will be inefficient because there will be too many temporary nodes to handle (one per each fault). This again prompts for a way to address faults and their effects with node numbers. The idea of a *fault-family* is introduced for this purpose.

Each circuit node can be set to one of two logical values: 1 or 0. When the fault-free value of a node is 1, then the faulty value is 0 and vice-versa. A fault has the effect of forcing a wire to a faulty value, which in turn will also affect some node value. When the wire containing a fault is the output wire of a gate, then forcing the value of the wire to a value is the same as forcing the value of the corresponding node to that value. On the other hand, if the affected wire is a fan-out wire, then it must be an input to some other logic gate, and the downstream node it affects will be the output node of that logic gate. Based on this observation, we can conclude that any fault has the primary effect that a node will be forced to an erroneous value, and so this primary effect can be described by the *node number* and this *value*.

The primary effect is not unique to each fault. For example, the primary effect of a stuck-at-1 fault affecting the output node of a NAND gate can be caused by a stuck-at-1 fault at the output wire or a stuck-at-0 fault at a gate input. We call a set of faults that have the same primary effect a *fault-family*. For a circuit with x nodes, there are $2x$ fault-families.

With fault-families, faults can be more easily inserted into the fault simulation on-the-fly. When we want to simulate a fault in parallel along with other faults, we can evaluate the gates until after the node that contains the primary effect of the fault being evaluated. At this point, the entire fault-family, which contains the effect of the fault, can be inserted instead of individual faults. Since the fault-family is node-based, this step avoids the fan-out wire problem we mentioned previously. A side effect of using fault-families rather than faults is that multiple faults can be simulated in one PE. Since each fault-family represents a number of faults, when more

NEED EVALUATION	NEED INSERTION	Gate Status
-	-	The gate is not currently affected by any fault
X	-	One or more fault-families propagate through this gate
-	X	The gate originates the primary effect of a fault-family
X	X	The gate is affected by more than one fault-family

Table 7.3: Interpretation of Flags in the Event-List

than one of these faults needs to be simulated, the fault-family can be inserted only once, thus freeing up PEs and thereby allowing more faults to be simulated in one pass.

Fault Insertion Using the Event Heap

Multiple fault insertion is done using the *event heap* introduced in Chapter 5. Recall that the event-heap is a random-in-sorted-out data structure used to ensure correct gate evaluation order. When each gate is evaluated, it's output value is checked against the expected good value. If the output node is erroneous, then all gates connecting to that node are added to the heap for further fault propagation. Accompanying this heap structure is an array that is used to make sure that each gate is added to the list once only in each pass. This array, named *event-list*, must be modified to support fault-parallel fault insertion.

Recall from Chapter 5 that the event-list was used for storing Boolean values, where a 1 indicates that a gate is already in the heap and a 0 otherwise. You can view this as a byte with bit number 0 indicating the Boolean *NEED_EVALUATION*. In *simf_fps*, this data structure is extended with bit 1 indicating a new Boolean *NEED_INSERTION*. During gate evaluations, when a gate with either bit set is popped (fetched) from the event heap, the following fault insertion action can be carried out.

```
initialize heap;
```

```
for all triggered fault-families do
{
    set NEED_INSERTION bit
    insert fault family into heap
}

while (j = pop heap)   // get the first element in the heap
{
    if (NEED_EVALUATION bit is set)
    {
        Evaluate_Gate (j);
    }

    propagation = false;

    if (NEED_INSERTION bit is set)
    {
        Insert_Fault_Family(j);
        propagation = true;
    }

    if (NEED_EVALUATION bit is set and
        NEED_INSERTION bit is not set)
    {
        if (output node is erroneous)
            propagation = true;
        else
            propagation = false;
    }

    if (propagation)
    {
        add fan-out gates to the heap
          with NEED_EVALUATION bit set;
    }
}
```

The above code fragment demonstrates the use of the bit flags. Before the next round of simulation starts, each fault-family is checked to see if it contains any member fault triggered by the test pattern. The *node number* of the fault-families that need to be simulated are added to the heap, and the corresponding byte in the event-list has the NEED_INSERTION bit set to 1. This is the *Prepare_Heap* procedure shown in Figure 7.1. Fault simulation may now begin. The first element in the heap (the one with the least gate number) is popped. If this gate needs to be evaluated, then

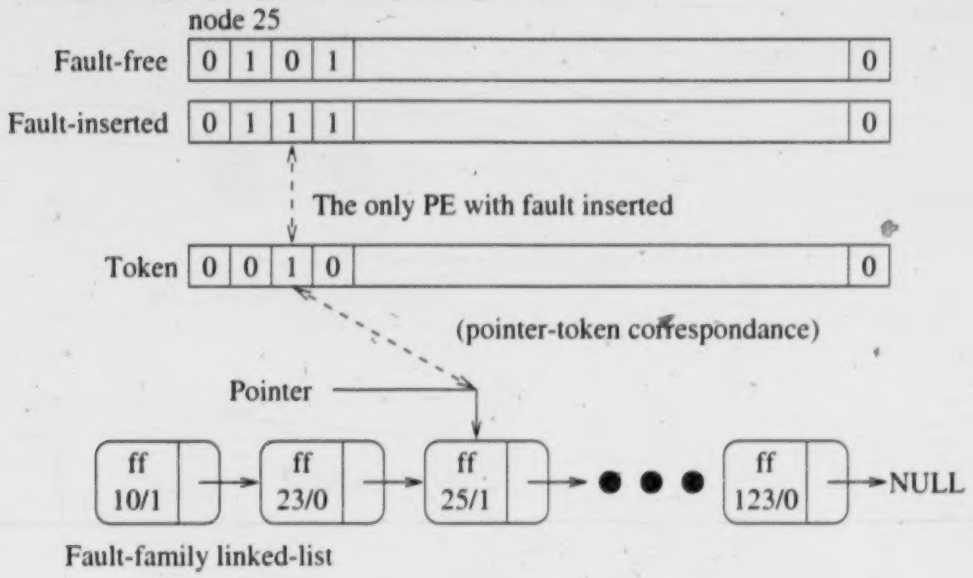


Figure 7.5: Fault Insertion With the Token-passing Method

a gate evaluation is carried out. Then the NEED_INSERTION bit is checked. If this flag is set, the fault will be inserted. Fault insertion can be done with a simple Exclusive-OR operation with one single PE active. How this is done will be explained in more detail in the next paragraph. If a fault insertion took place, then it is certain that the fan-out gates of the current gate need to be evaluated so that the effect of the inserted fault can be propagated. Otherwise, the node value is checked against the fault-free value. If the node is erroneous, meaning that it is affected by a fault, then fault propagation must also be performed. Finally, if fault-propagation is needed, the fan-out gates will be added to the heap with the NEED_EVALUATION bit set. These fan-out gates may already be in the heap with the NEED_INSERTION bit set; in which case, both bits will be set for the gate. When such a gate is popped from the heap, both gate-evaluation and fault-insertion will be done. Table 7.3 describes these flags and their implication.

When C•RAM is used to implement the fault-parallel fault simulation algorithm, each PE represents a fault-family. When a fault is inserted, the corresponding fault-family is inserted into one PE only. Given a fault-family, we need to know which PE is representing it before fault-insertion can take place. One way to do this is to

associate each fault-family with a *PEid* so that a PE can be selected when desired. Selecting one arbitrary PE out of all the PEs, however, is a very time consuming process[5]. In order to reduce this overhead, a token-passing method is used. Before fault simulation, the fault-families needing to be inserted are stored in a list in order of ascending gate number, which will be the same as the order in which they will be inserted during fault simulation. A pointer is initialized to point to the beginning of this list. In addition, a **cboolean** variable is initialized with all bits set to 0 except for the first bit (PE0), which will be used as the first instance of the *token*. During fault simulation, when fault-insertion is encountered during the fault simulation loop, the PE with the token is used to represent the selected fault-family. After this fault-insertion, the pointer is advanced to point to the next fault-family in the list; this is accomplished simply by shifting the **cboolean** variable to the right by one bit. This procedure is repeated until all the gates in the event-heap are evaluated (and fault inserted). Figure 7.5 illustrates this token-passing method.

Fault Dropping

Fault detection and fault dropping are done after each pass of fault simulation. Recall that when more fault-families need to be simulated than the number of PEs, more than one pass is also needed: i.e. for each test pattern being simulated, the fault dropping procedure may be called more than once. The procedure is called multiple times so that the C•RAM can be cleared for the next pass.

After the current pass of fault-simulation, C•RAM will contain a set of **cbooleans** representing the state of the primary outputs of the faulty circuits that were simulated in parallel. At the beginning of fault-dropping, the values of these **cbooleans** are compared to the fault-free values, and the result of this comparison is collected into one single **cboolean**, where a 1 will indicate a PE with a detected fault and 0 otherwise. Finally, role-calling is used to determine which fault-family is actually detected. *Role-calling* is the process where the corresponding PEid is returned when a PE contains a "dirty" bit in a specified **cboolean** [7]. When role-calling is first called with a pointer to the **cboolean** variable in question, $O(log_2 n)$ binary search

[5] $O(log_2 n)$ C•RAM operations are needed for each selection, where n is the number of PEs in total.

Circuit	simf time	PPS time	simf/PPS speed-up	FPS time	simf/FPS speed-up
c1355nr	0.641s	0.190s	3.37	1.756s	0.36
c1908nr	1.497s	0.259s	5.77	12.615s	0.12
c2670nr	68.021s	6.675s	10.19	1365.713s	0.05
c3540nr	1.689s	0.358s	4.71	12.956s	0.13
c5315nr	2.050s	0.443s	4.63	14.415s	0.14
c6288nr	9.447s	1.730s	5.46	3.141s	3.01
c7552nr	76.836s	9.170s	8.38	3971.439s	0.02

Table 7.4: Result Comparison of PPS with FPS

is carried out, and the PEid of the first PE that contains a 1 in the `cboolean` is returned. In subsequent calls, the next PE that contains a 1 will return. When no more PEs contain a 1, -1 is returned as the PE number. In this way all PEs can be polled efficiently.

In our implementation, an extra feature is added to further speed up the role-calling process. Before going into a binary search, the function first considers the PE next to the right of the PE returned in the previous call. If this PE also contains a 1 in the `cboolean`, then the PEid of this PE is returned without going into the binary search. In the worst case, where the `cboolean` contains a bit pattern such as 01010101010101...., this heuristic is slower. However, in most of the cases, especially when the 1's are often clustered, this method should yield good performance. We found that 1's were often clustered in this way for our benchmark fault simulations when the fault coverage is low.

For each of the PE numbers returned by role-calling, the fault dropping routine will search for the corresponding fault-family. Each member fault of the fault-family is again checked against the triggering condition. If the member is triggered, the fault and the faults implied by it are marked as detected. The detected faults are then dropped from the fault list. If the fault-family does not contain any more undetected faults, it will be dropped from the fault-family list.

7.3 Evaluation

The fault-parallel fault simulation algorithm is more complicated than the pattern-parallel algorithm. A large portion of the program uses only the CPU to run (such as fault-free simulation and heap initialization and loading). Many subroutines use both the C•RAM and the CPU (such as fault simulation and fault dropping). In order to evaluate the speed-up of the simulator, two measurements were taken: the amount of CPU time needed to run fault-free simulation, heap preparation and the other sequential processes common to the other simulators, plus the C•RAM run time. It is assumed that the C•RAM run time is adequate to represent the run time of those subroutines that are dominated by C•RAM operations.

7.3.1 Test Results

Table 7.4 compares the results of pattern-parallel fault simulation (PPS) with those of fault-parallel fault simulation (FPS). It can be seen that in most of the cases, simf_fps is worse than the non-C•RAM version (with speed-up ratios of less than 1). On the other hand, *simf_pps* is at least 4 times faster than *simf*. One interesting exception, however, is the c6288nr circuit. It is the only case that runs faster than the conventional version.

It should be noted that only 256 test patterns were simulated for the c6288nr circuit. This is because the circuit is so easy to test that roughly that many patterns are needed to achieve more than 99% fault coverage. For all the other circuits, at least 1024 test patterns were simulated. For the two "hard" benchmark circuits, namely c2670nr and c7552nr, 256k test patterns were used. The results shown in Table 7.4 here agree with our theory, stated in Chapter 4, that fault-parallel fault simulation is useful when there are a lot of undetected faults.

Another reason for the disappointing performance of *simf_fps* is due to inefficient fault-free simulation. In Table 7.5, the simulation times of *simf_fps* are broken down into two pieces: the CPU portion and the C•RAM portion. It is shown that more than 90% of the simulation time is spent in the CPU. Part of the CPU time is used for fault-free simulation (from 40% to 80% in Table 7.8). This is because only one pattern is evaluated during fault-simulation. A possible way to solve this problem

Circuit	patterns	C•RAMtime	CPU time
c1355nr	1024	0.035s(2.0%)	1.721s(98.0%)
c1908nr	3328	0.128s(1.0%)	12.487s(99.0%)
c2670nr	256k	20.165s(1.5%)	1345.548s(98.5%)
c3540nr	1536	0.212s(1.6%)	12.744s(98.4%)
c5315nr	1024	0.278s(1.9%)	14.136s(98.1%)
c6288nr	256	0.083s(2.7%)	3.058s(97.3%)
c7552nr	256k	8.873s(0.2%)	3962.566s(99.8%)

Table 7.5: Analysis of the Fault-Parallel Fault Simulation Results

Circuit	Num. FF sim	Num. PE used	PE util. FPS	PE util. PPS	Ratio FPS/PPS	Number of patterns
c1355nr	16k	1039k	1.55%	1.54%	1.0	1024
c1908nr	68k	3472k	1.96%	6.03%	0.32	3328
c2670nr	19642k	287815k	6.83%	22.17%	0.31	256k
c3540nr	132k	1703k	7.77%	3.65%	2.1	1536
c5315nr	185k	1232k	14.98%	2.56%	5.9	1024
c6288nr	24k	286k	8.52%	0.23%	37	256
c7552nr	7783k	275956k	2.82%	19.28%	0.15	256k

Table 7.6: PE Utilization : Average Number of FFs Simulated Per Pass

is to better utilize the CPU's bit-parallel datapath to simulate 32 test patterns at once. Another possible way is to use the C•RAM to simulate many test patterns in parallel. Either of these two methods suggests the use of a hybrid scheme combining the pattern-parallel and fault-parallel approaches.

In the previous chapter, we evaluated *simf_pps* using the PE utilization measure. In *simf_fps*, the corresponding PE utilization is measured by calculating the average number of fault-families simulated per simulation pass. The results are shown in Table 7.6. We see that PE utilization for *simf_fps* is very close to that of *simf_pps*. For the circuits simulated with more patterns (c1908nr, c2670nr, c7552nr), PE utilization in fault-parallel fault simulation is clearly less than that of pattern-parallel fault simulation. For the easy circuits with not so many patterns simulated (c1355nr, c5315nr, c6288nr), the PE utilization ratio in fault-parallel fault simulation is up to 37 times greater than that of pattern-parallel algorithm. Once again our results give us strong

Circuit	Without DFS	With DFS	Speed-up	Speed-up (C•RAM)
c1355nr	1.756s	2.025s	0.87	1.00
c1908nr	12.615s	13.913s	0.91	1.00
c2670nr	1365.713s	1469.902s	0.93	1.00
c3540nr	12.956s	13.867s	0.93	1.00
c5315nr	14.415s	15.081s	0.96	1.00
c6288nr	3.141s	3.321s	0.95	1.01
c7552nr	3971.439s	4040.630s	0.98	1.01

Table 7.7: Effect of Depth-First-Search from Output Optimization

support to justify the development of a dynamic hybrid fault simulation method to take advantage of the good PE utilization at the beginning of the simulation, when there are many easy-to-detect faults remaining.

7.3.2 Optimization with Fault Grouping

In [24], the *PROOFS* fault simulator was described. PROOFS uses the 32-bit datapath in the CPU to perform fault-parallel fault simulation. In addition to the basic design of the simulator, the authors also describe a way to optimize simulation speed, called the *fault grouping* technique. It was found that fewer gate evaluations are required when faults are grouped using a *depth-first-search from the primary outputs*. Two versions of the *simf_fps* simulator were designed: one with the fault grouping technique and one without. The results are shown in Table 7.7.

We can see from the table that using the fault grouping technique does not greatly reduce simulation time (only a slight improvement was obtained for c6288nr and c7552nr). Moreover, it takes up more CPU time. The primary advantage of fault grouping is that faults (or fault-families) with similar output stems will be put closer together so that the benefits of simulating them together can be maximized (by propagating the faults to fewer gates). In PROOFS, when there are thousands of faults with only 32-bit parallelism, the simulator will have to run many passes to finish fault simulating a single test pattern. Having groups of 32 faults with similar output stems is easily achievable, and it is therefore a good technique to use in PROOFS. In *simf_fps*, however, many more faults are simulated in parallel with less

Circuit	Patterns	CPU	Fault-free operations	Fault Triggering operations
c1355nr	1024	0.782	0.300(38.4%)	0.481(61.6%)
c1908nr	3328	4.412	2.241(50.8%)	2.172(49.2%)
c2670nr	256k	356.684	202.437(56.8%)	154.247(43.2%)
c3540nr	1536	4.229	1.831(43.3%)	2.398(56.7%)
c5315nr	1024	4.532	1.804(39.8%)	2.729(60.2%)
c6288nr	256	0.900	0.409(45.4%)	0.492(54.6%)
c7552nr	256k	798.324	636.811(79.8%)	161.513(20.2%)

Table 7.8: Fault Triggering Detection Time in FPS

passes per pattern, and the benchmark circuits are not very big. The achievable speed improvement is therefore insignificant. Although the advantage of fault grouping is not obvious here, it may be useful for the hybrid fault simulator, which will be discussed in the next chapter.

7.4 Discussion

A very obvious bottleneck of the fault-parallel fault simulator is the fault-free simulation. As mentioned above, we could improve the simulator's speed by simulating the fault-free circuits in a pattern-parallel fashion. Another bottleneck, which is also caused by single pattern fault-free simulation, is fault-triggering detection, which was not discussed in the previous sections. In the pattern-parallel fault simulators, not only is simulation (both fault-free and fault inserted) done in parallel, so is fault triggering detection. For a simulation with 256k test patterns using *simf_pps*, each fault is checked at most 256 times for fault triggering detection. However, in *simf_fps*, this process has to be done for each test pattern, which means up to 256k such checks have to take place. From Table 7.8, we can see that the fault triggering detection time can take up as much as 60% of the CPU run time. One possible solution to remove the detection bottleneck is not to perform fault triggering detection at all, which would involve redesigning part of the fault simulation engine.

Another observation can be made regarding fault implication. Fault implication was mentioned in Chapter 3 and was implemented in both *simf* and *simf_pps*. Speed

improvement was achieved in these implementations. In *simf_fps*, fault implication is realized among the members within a fault-family. However, this strategy does not reduce simulation effort as much as it does in the pattern-parallel fault simulators. One reason is that the list of fault-families was generated once only even if a test pattern requires multiple passes. After each pass when faults are dropped from the fault list, some fault-families in the next pass may no longer need to be simulated due to detection via fault implication. However, there's no mechanism right now to take advantage of this feature. It is not clear at this point whether an implementation of better fault implication can significantly speed-up the fault-parallel fault simulation. It is thus another interesting issue for future research.

Chapter 8

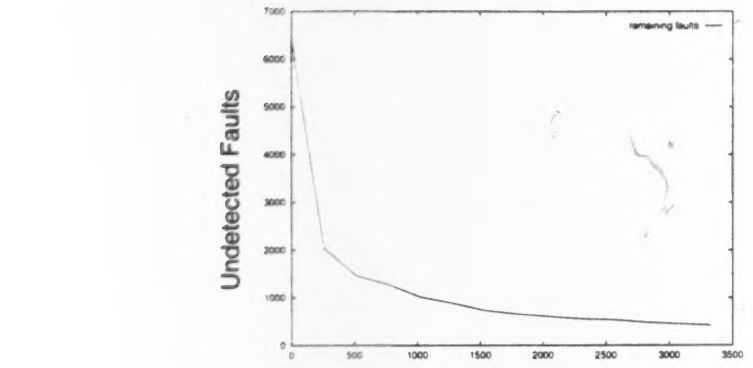
Hybrid Fault Simulation on C•RAM

8.1 Overview

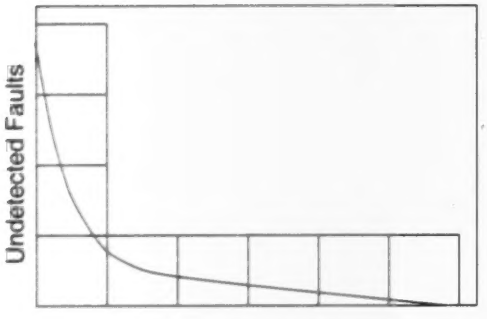
In Chapter 6, we described a simulator that implements pattern-parallel fault simulation using PPSFP as the basic algorithm. It was found that this simulator is generally efficient, although it doesn't always utilize C•RAM PEs very well. In Chapter 7, a fault-parallel fault simulator was described. The simulator is generally too slow because of the inefficient fault-free simulation process and fault triggering detection. Nevertheless, the simulator demonstrated that under certain conditions it can run faster than the pattern-parallel fault simulator due to relatively high achievable PE utilization. A hybrid of these two simulators will be described in this chapter. We called it *simf_hps* for Hybrid-Parallel Fault Simulator.

The hybrid-parallel fault simulation scheme can be further categorized into two different schemes: *static-hybrid fault simulation* and *dynamic-hybrid fault simulation*. A static-hybrid fault simulation scheme is one where the *fault-to-pattern ratio*[1] in each round of a simulation remains unchanged throughout the fault simulation of the whole circuit. On the other hand, a dynamic scheme is one where the fault-to-pattern ratio in each simulation pass may be adjusted from one simulation round to another. The difference between the two schemes is illustrated in Figure 8.1. Each chart here contains a typical fault dropping curve. The little tiles representing fault-simulation rounds must be used to cover up the area under the curves. The area under the curve represents the lower bound on the number of pattern-fault pairs that

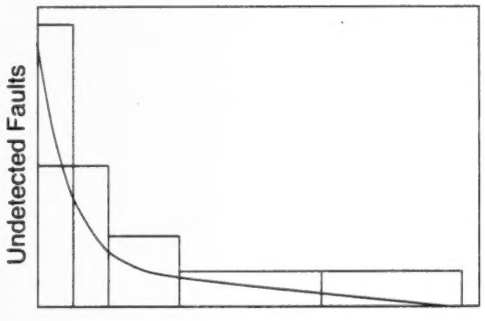
[1] Number of faults simulated in parallel versus number of patterns simulated in parallel.



(a) Fault Dropping Curve of c1908nr



(b) Static Hybrid (fixed tiles)



(c) Dynamic Hybrid (adaptive tiles)

Figure 8.1: Static-Hybrid Fault Simulation versus Dynamic-Hybrid Fault Simulation

must be simulated. All of the tiles have the same area, while the different shapes mean different fault-to-pattern ratios. The curve in Figure 8.1(b) is covered by tiles of the same shape representing a static-hybrid fault simulation, whereas the one in Figure 8.1(c) represents a dynamic one using tiles with different shapes. The static method requires 9 tiles to complete the task, while the dynamic one needs only 6 tiles; this implies that the dynamic scheme is sped up 1.5 times with respect to the static scheme since 3 fewer fault-simulation rounds are needed. The simpler static-hybrid method was implemented first in this research. The dynamic-hybrid fault simulator is a modification of the static-hybrid fault simulator with an additional routine that adjusts the f (number of faults simulated in parallel) and p (number of patterns

simulated in parallel) values.

8.2 Design Details

Figure 8.2 illustrates the flow of a fault-simulation round that applies for both static and dynamic hybrid-parallel fault simulation. After the next set of patterns has been read and fault-free simulated in C•RAM, a new fault-simulation round begins. In each round, the program tries to fill up a *simulation queue* for simulation. The simulation queue stores pointers to the *fault-groups* that will be simulated in the next pass. When a queue is filled up, fault simulation begins. The process of fault simulation includes both fault evaluation and fault insertion, as described in the previous chapter. At the end of a round, the node values at the primary outputs are compared to the corresponding fault-free values. Each fault-group in the simulation queue is then checked for detected faults. If all the faults in the fault-group have been detected, then that fault-group is deleted from the linked-list of fault groups. A fault-simulation round ends after simulating all the triggered fault-groups. In a dynamic-hybrid fault simulation, an additional step for adjusting the fault-to-pattern ratio can be added as shown in the flow-chart with dashed-lines.

8.2.1 Partitioning C•RAM Memory

The first important concept for implementing hybrid-parallel fault simulators on C•RAM is *C•RAM partitioning*. We can logically divide the C•RAM into P partitions, each with the same number of PEs, so that different partitions can handle slightly different jobs. For simplicity, the number of PEs in C•RAM is assumed to be a power of 2. In the diagram shown in Figure 8.3, the C•RAM is logically divided into $p = 4$ partitions. In order to provide access to individual partitions, a *partition_mask* C•RAM variable is introduced. The *partition_mask* variable is basically a `cint` with the number of bits equal to P, the number of partitions. Each row of the variable is a mask of the corresponding partition. In the diagram shown, the black stripe represents a row of 1s. With this *partition_mask* set up, we can easily select any combination of the partitions for separate operations. Another C•RAM variable used in this program, which is related to the C•RAM partitions, is the *PE_mask*. It

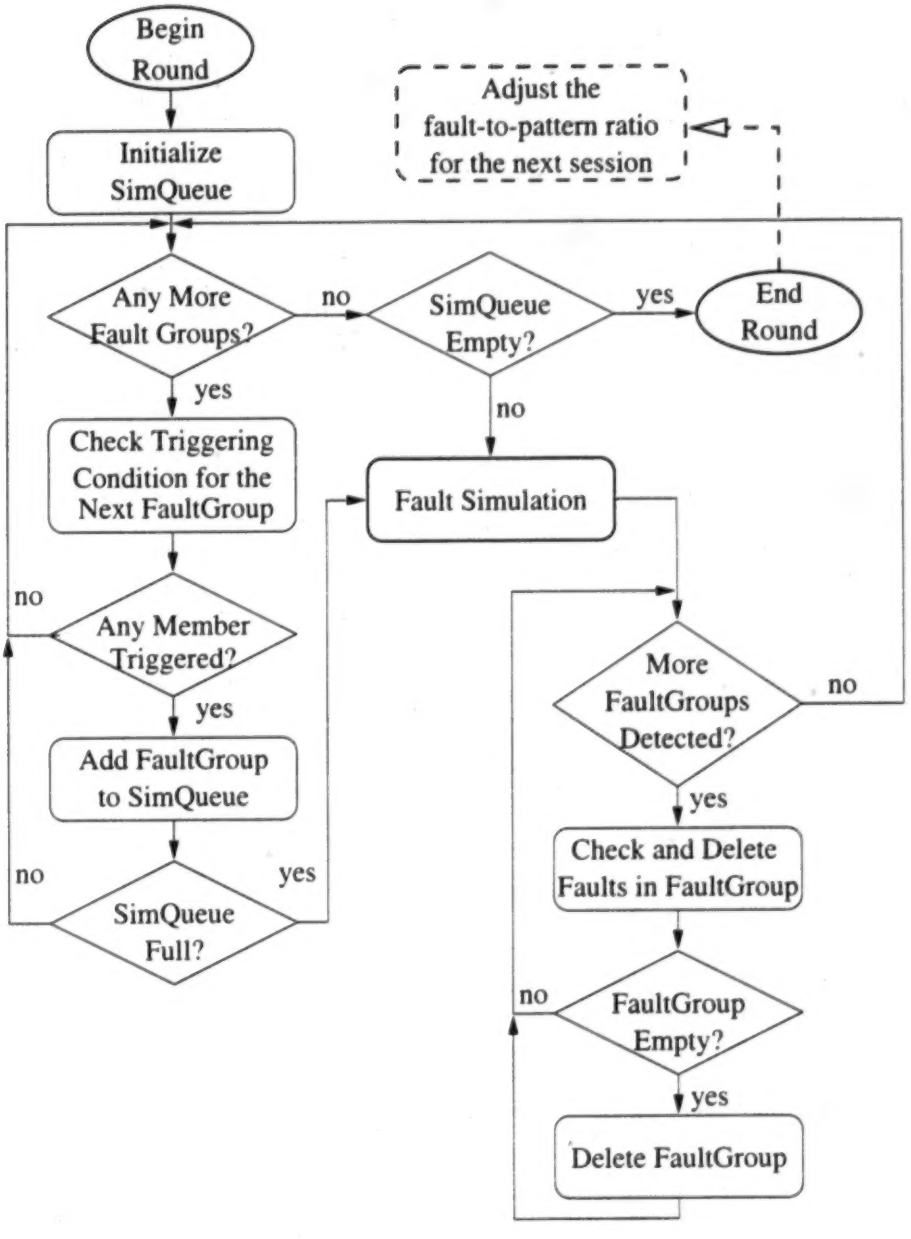


Figure 8.2: Flow Chart of a Fault-Simulation Round

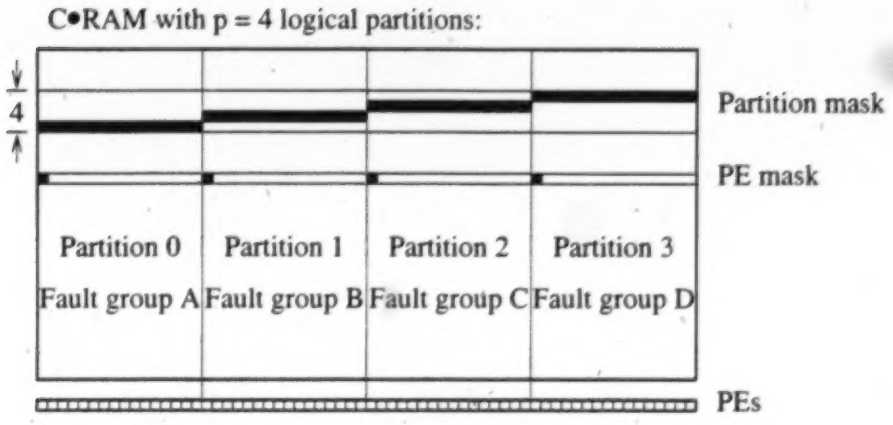


Figure 8.3: Logical Partition of C•RAM

allows the selection of one PE out of each partition.

In fault-simulation, there are at least two ways to use the partitions. One way is to have each partition perform fault-parallel fault simulation with one particular test pattern. We call this the *fault-oriented method* because it is similar to purely fault-parallel fault simulation. The other way is to have each partition simulating the same set of test patterns and one particular fault. This we call the *pattern-oriented method*. We chose to implement the second method for three reasons: First, since the *pattern-oriented method* resembles pattern-parallel fault simulation, the design is much more elegant and more easily understood. Second, fault triggering for transition faults involves comparing the current and previous test patterns. In *simf_pps*, a method was invented so that only one shift in C•RAM is needed to perform the comparison, whereas in *simf_fps*, a lot more C•RAM memory copying is required. Finally, we have already seen that *simf_pps* can outperform *simf_fps* in most cases.

In a static-hybrid fault simulation, C•RAM partitioning is done only once at the beginning of the program. On the other hand, a dynamic-hybrid fault simulator can adjust the fault-to-pattern ratio by changing the number of partitions. This can be easily done by calling a *SetPartition* routine.

The *SetPartition* routine sets up both the *partition_mask* and the *PE_mask* variables in C•RAM. Both variables can be implemented using the *PEid* feature of the C•RAM. PEid is a `cint` variable that holds a different value (identity) for each PE. This value is simply the array index of the PEs ($0 \,..\, n$).

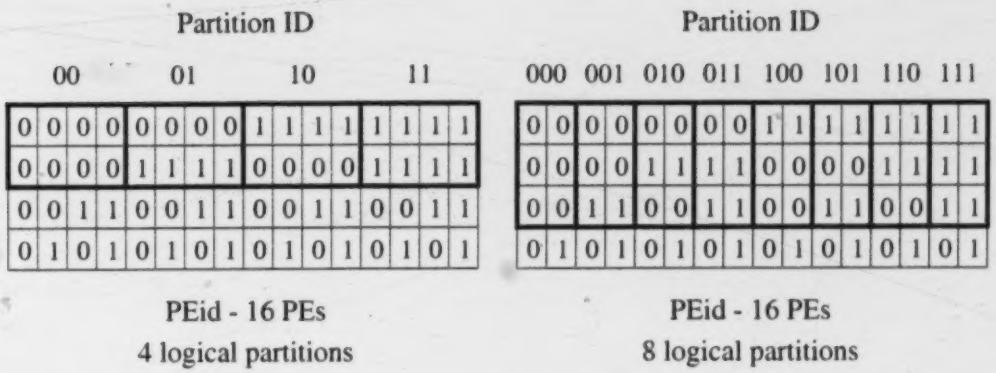
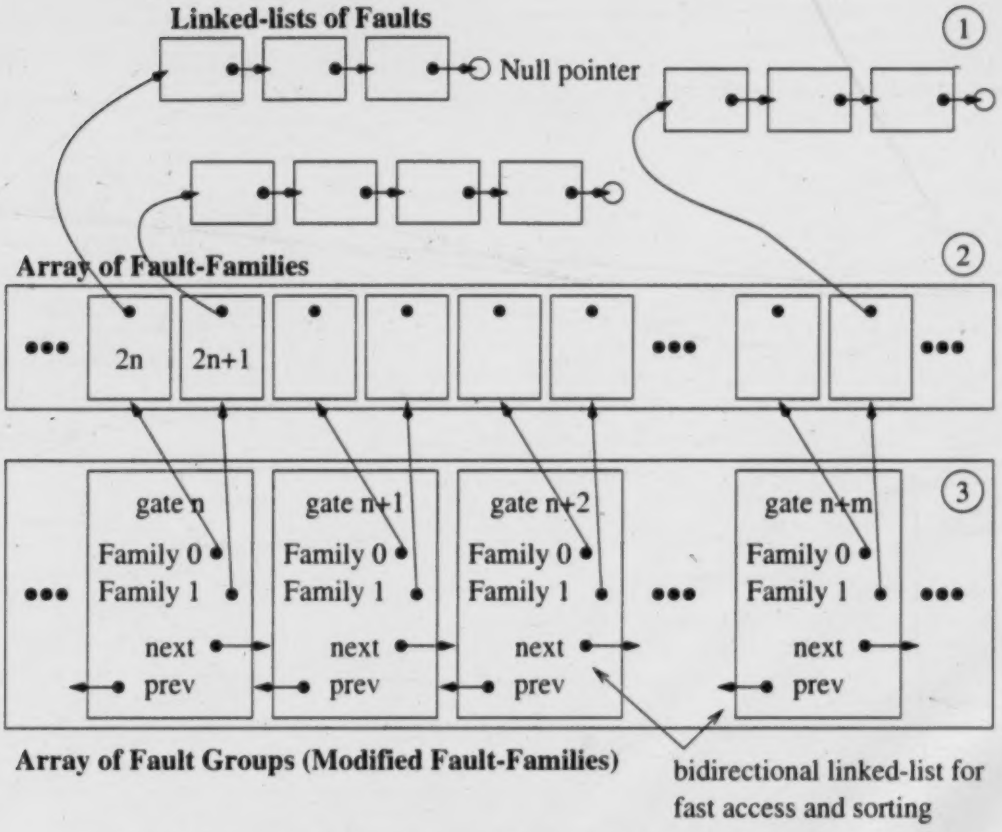


Figure 8.4: Partitioning With PEid

When creating the *partition_mask*, note that the mask can be created using the most significant bits of the PEid variable, which is unique to each PE. Figure 8.4 illustrates a C•RAM with 16 PEs divided into 4 logical partitions and 8 logical partitions. It can be seen that, in each case, the M most significant bits ($M = log_2 P$) of the *PEid* variable are the same for each PE in the same partition, which can therefore be used as the corresponding partition's *partition_ID*. Each partition's mask can thus be constructed in M steps. The *PE_mask* can be created in similar fashion. However, a much faster method is to recognize that there is a row of bits in PEid, namely the one corresponding to the least significant bit of the *partition_ID*, which can be used to set up the *PE_mask* in three steps as follows:

```
PEid.operate (PEid.bits-nMSD, op_mbar, writex); // load a bit in PEid
PE_mask.operate (0, op_x, shiftright);          // shift bits to right
PE_mask.operate (0, op_x ^ op_y, groupwrite);   // exclusive or
```

The first step loads the interesting row of bits inverted into the x-register. Then, the bits are shifted to the right and written to the y-register. Here we assumed that '0' is going to be shifted in from the left at the leftmost PE. Finally, the two registers are exclusively OR-ed to get the mask. This optimization does not affect the C•RAM run-time very much because it's not a highly repeated step. Nevertheless, we described it here for future reference.



1: Data structure used in pattern-parallel fault simulation (as a singly linked-list)
1+2: Data structure used in fault-parallel fault simulation
1+2+3: Data structure used in hybrid-parallel fault simulation

Figure 8.5: Data Structures Supporting Fault Grouping

8.2.2 Modified Fault-Family Fault Simulation

The pattern-parallel fault simulator used a simple fault list data structure. In fault-parallel fault simulation, a more complicated data structure called a fault-family was implemented, as described in the last chapter. Yet another data structure is required for hybrid-parallel fault simulation. The new structure, which we call a *fault group*, is a modification of the fault-family data structure. Note that this is different from the fault-group mentioned in the PROOFS paper [24].

In *simf_fps*, only one test pattern was simulated in a pass. In that case, only one of the two fault-families associated with a node will be inserted for simulation at any

Circuit	HPS	HPS + DFS	Speed-up
c1355nr	0.111s	0.108s	1.03
c1908nr	0.239s	0.221s	1.08
c2670nr	12.205s	11.708s	1.04
c3540nr	0.426s	0.390s	1.09
c5315nr	0.488s	0.466s	1.05
c6288nr	0.562s	0.525s	1.07
c7552nr	20.883s	20.636s	1.01

Table 8.1: Speeding-up HPS by Using Depth-First-Search Grouping

time because one test pattern cannot set a node value to both 1 and 0. This is not the case in hybrid-parallel fault simulation. Since more than one pattern is simulated in each pass, a node value can be set to both 0 and 1 in the same simulation pass by different test patterns, which in turn can possibly trigger faults in both fault-families of the same node. Wouldn't it be nice to simulate both fault-families in the same partition, saving up one partition for other faults? The fault-group data structure was primarily designed for this purpose. A complete view of the data structure hierarchy is shown in Figure 8.5.

At the outermost level (level 3), an array of fault-groups is allocated with the same number of elements as the number of gates. Each fault-group is associated with a gate number. The faults represented by a fault-group are all the faults that have *primary effects* on the output of that gate. These faults are organized into two fault-families in each fault-group. One family contains all the faults that will force the gate's output to 0, and the other with all the faults that force that output to 1. With this data structure, all the faults with primary effect on a gate can be found very efficiently given a gate number.

The array of fault-groups is further organized by a bi-directional linked-list. This structure is useful for deleting detected fault-groups efficiently after fault simulation. It also provides a way to sort the fault-groups in an order different from the gate-evaluation order, allowing the use of optimization methods such as the depth-first-search from output method described in [24]. The simulator was implemented with a switch to enable or disable the depth-first-search gate ordering method.

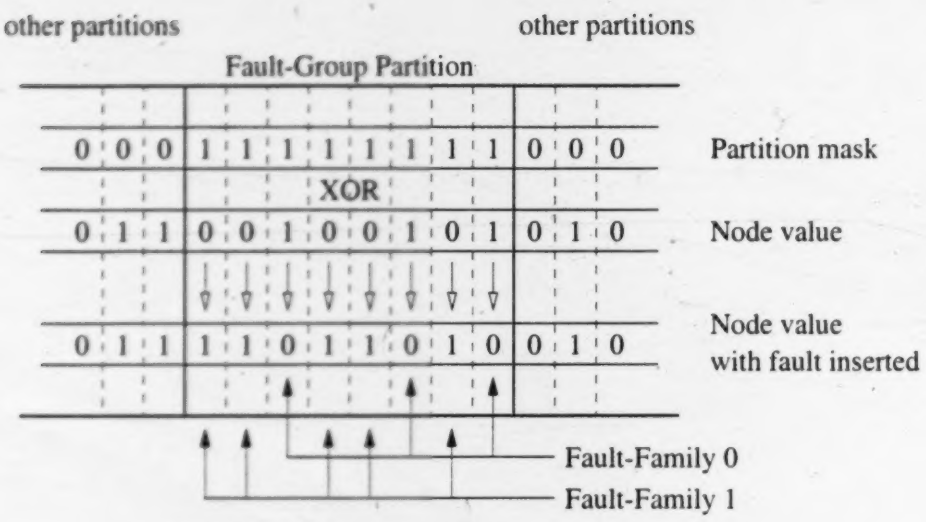

other partitions
other partitions
Fault-Group Partition
0 0 0 1 1 1 1 1 1 1 1 0 0 0
Partition mask
XOR
0 1 1 0 0 1 0 0 1 0 1 0 1 0
Node value
Node value
with fault inserted
0 1 1 1 1 0 1 1 0 1 0 0 1 0
Fault-Family 0
Fault-Family 1

Figure 8.6: Simulating With Fault-Groups in a C•RAM Partition

The method of applying depth-first-search from the primary outputs to sort gates was introduced in the *PROOFS* paper [24]. When we applied it in the fault-parallel fault simulator, there was no significant advantage. We concluded that the method should only be applied when the number faults simulated in parallel is relatively small. Table 8.1 shows that this method can improve the performance of the static-hybrid fault simulator slightly, most of the time. The results were obtained with four C•RAM partitions. The depth-first-search method was therefore used in all of our hybrid fault simulators.

Figure 8.6 shows a C•RAM partition with a fault-group inserted for fault simulation. In this diagram, we can see that a C•RAM partition is capable of simulating the two fault-families associated with the fault-group at the same time. The primary effect of the faults associated with the fault-group can be simulated by simply inverting all the bits that gives the node value represented by the fault-group. The *partition_mask* is used here to select only the partition simulating the fault-group affected by the inserting operation (simply Exclusive-OR the *partition_mask* and the original node value).

Circuit	HPS	HPS with C•RAMMapper	Speed-up
c1355nr	0.111s	0.104s	1.07
c1908nr	0.239s	0.225s	1.06
c2670nr	12.205s	10.060s	1.21
c3540nr	0.426s	0.381s	1.12
c5315nr	0.488s	0.445s	1.10
c6288nr	0.562s	0.501s	1.12
c7552nr	20.883s	19.341s	1.08

Table 8.2: Speeding-up HPS by Using C•RAM Mapper

8.2.3 Hybrid-Parallel Fault Simulation

The fault simulation process is very similar to fault-parallel fault simulation described in Chapter 7. Before fault simulation takes place, each fault-group is checked to see if it's triggered. If it's triggered, it is added into the *simulation queue*, which is basically an array of pointers into the array of fault groups The number of fault-groups in the queue is kept in a counter.

As fault-groups are added to the simulation queue, the gate number of the group is also pushed into the sorted *event heap* and the *NEED_INSERTION* flag is set at the corresponding location in the event list array. When fault simulation is finally carried out, gate evaluation and fault insertion are based on the flags in the event list, following the same procedure described on page 93.

There are two implementation options for the fault simulation data structure. One option is to use a structure similar to that used for pattern-parallel fault simulation, with a large `cint` variable whose size in bits equals the number of gates in the circuit. This method is easy to implement, but a lot of copying must be done to keep the simulation `cint` (*wire_value*) synchronized with the `cint` containing the fault-free values (*good_value*). A second option is to use the C•RAM mapper, which allocates and frees `cbooleans` whenever necessary. This option can save copying time, however it is more complicated to implement. Both options were implemented for this simulator, and the difference is shown in Table 8.2. We can clearly see that using C•RAM mapper is more efficient. Since both *simf* and *simf_pps* implement the first option, the hybrid fault simulator has the first option implemented so that a fair comparison

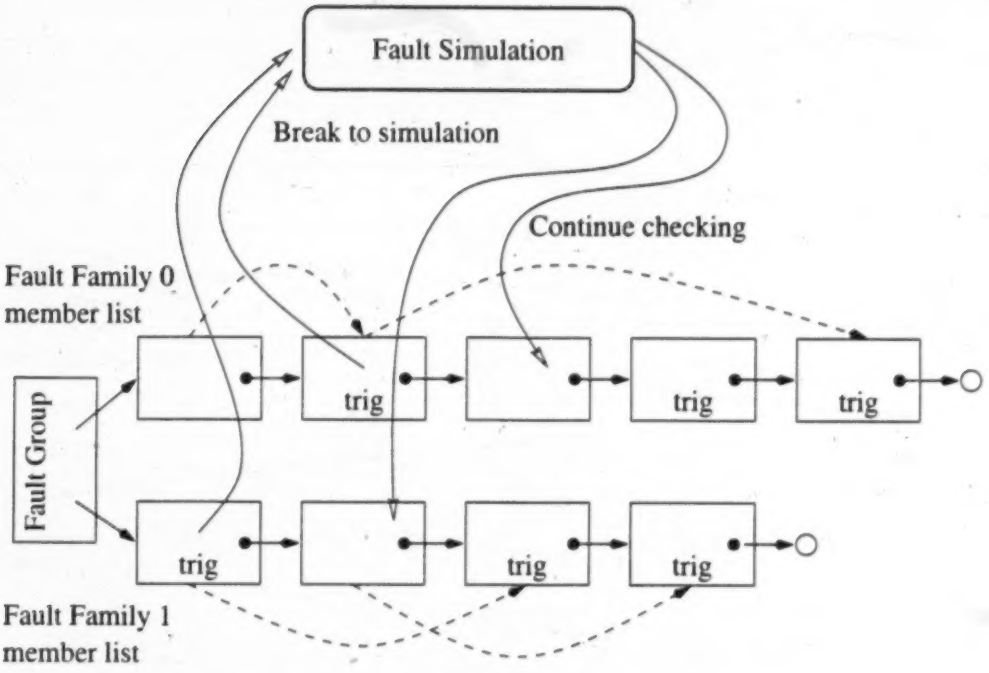


Figure 8.7: Fault Triggering/Detection Mechanism

between the three simulators could be carried out. The results are discussed later in Section 8.3.

8.2.4 Fault Triggering and Fault Detection

When the fault triggering and fault detection mechanisms were designed, the primary goal was to reduce the amount of unnecessary checking as much as possible. A design was used that allows the fault-group to branch to the fault simulation routine as soon as it is determined that some faults in the group are triggered, a design that has the ability to resume checking from the fault after the fault that got triggered during the simulation. This is shown in Figure 8.7. Here, the member list of each of the fault-families is traversed to check for triggering conditions. Within each list, if a triggered fault is encountered, the traversing is stopped and the fault's position is saved. The fault-group can then be added to the simulation queue. After fault simulation, scanning through the member list picks up again at the triggered fault. In this second phase, all triggered faults (including the first one) will be deleted from

the list of faults for the current fault group. With this type of mechanism, each fault in the fault list is visited at most once. Our design minimizes wasted triggering detection effort or wasted fault simulation effort.

To implement our design, three routines were required, namely, *Triggered*, *DetectFault*, and *CreateMask*. *Triggered* and *DetectFault* each perform one pass along the current fault list. *CreateMask* is a Boolean function that takes a fault as input parameter. If the fault is triggered, a *true* value is returned, otherwise a *false* value is returned. In addition, a `cboolean` mask is created. If a pattern is capable of triggering the fault, the corresponding PE bit in the mask is set to 1. This function is used in both *Triggered* and *DetectFault* for checking the triggering condition of faults. When a fault is triggered, both the position of the fault in the fault list and the corresponding `cboolean` mask are saved. The pseudo code for this implementation is as follows:

```
while (current fault-group is not NULL)
{
    initialize empty simulation queue;

    while (simulation queue not full)
    {
        if (Triggered (current fault-group))
        {
            append current fault-group to simulation queue;
            advance to next fault-group;
            if (current fault-group is NULL) break;
        }
    }

    if (simulation queue is empty) break;

    perform fault simulation;

    for each fault-group in simulation queue
    {
        DetectFault (fault-group);
    }
}
```

After fault simulation, *DetectFault* is called to continue the checking. The saved `cboolean` mask is first checked against the simulation *result* with the logical XOR

operation, where *result* is also a `cboolean`. Bits in *result* are set to 1 if and only if the fault-free and faulty output values differ. The fault list traversal then continues with additional steps that check for fault detection and that drop faults from the fault list. *Mark_Detected* (page 67), which takes advantage of fault implication relationships, is used to accelerate the fault dropping performance.

Since we use the fault group data structure, which collects multiple faults and represents them with their primary effects, to represent faults, it is necessary to make sure that the primary effect of a fault is triggered before fault simulating it. Primary effect checking is necessary when the site of the fault is one of the input wires of a gate with multiple inputs, and that the wire is a fan-out wire. All of these checkings are done in the *CreateMask* routine.

8.2.5 Dynamic-Hybrid Fault Simulation

In order to design a dynamic-hybrid fault simulator, one must define a function for determining the fault-to-pattern ratio. There is an example of such function in [18]. From this example, we can see that it is important to find a *cost function* that can be used to accurately estimate the run time. The cost function that we developed for our hybrid fault simulation algorithm can be stated as follows:

$$\begin{aligned} Cost\ per\ Round &= GST + CMT * n_{undetected} + \frac{FST * n_{triggered}}{f} \\ GST &= Good\ Simulation\ Time \\ CMT &= CreateMask\ Time \\ FST &= Faulty\ Simulation\ Time \\ n_{undetected} &= number\ of\ undetected\ faults \\ n_{triggered} &= number\ of\ triggered\ faults \\ f &= number\ of\ faults\ simulated\ in\ parallel \end{aligned} \tag{8.1}$$

Where cost is in terms of the number of C•RAM clock cycles. The Good Simulation Time is circuit dependent. It is proportional to the total number of gate inputs (n_{input}) plus the total number of gates (n_{gate}). It is quite hard to obtain an exact CreateMask Time. However, by analyzing the source code of the CreateMask routine, it can be estimated as

$$\begin{aligned} CMT &= avg(t_{trigger}) + avg(t_{peffect}) \\ &= avg(t_{trigger}) + (avg(t_{eval}) + t_{setup}) \\ &= 3op + (n_{input}/n_{gate} + 1op) + 8op \end{aligned} \tag{8.2}$$

Where 3 and 8 (op = C•RAM operations) are the values of average cost for fault triggering ($avg(t_{trigger})$) and the setup cost of a gate evaluation (t_{setup}), respectively on our UltraSPARC 5 workstation. The ratio n_{input}/n_{gate} represents the average number of gate inputs per gate. Note that the above equation does not take into account the fact that some faults do not need a gate evaluation for checking the primary effects. This can be estimated by multiplying the average number of C•RAM clock cycles for primary effect checking by the fraction of faults ($r_{peffect}$) that need primary effect checking (or, the fraction of wires that satisfy the primary effect checking conditions).

Both GST and CMT can be determined before fault simulation takes place. The Faulty Simulation Time per pass (FST) however, depends on the nature of the faults being simulated. This is because fault simulation only evaluates the gates affected by the faults. We used the following approximation:

$$FST = \frac{number\ of\ gate\ evaluations\ in\ fault\ simulation\ in\ previous\ run}{number\ of\ passes\ in\ the\ previous\ run} \tag{8.3}$$

With GST, CMT, and FST defined as above, Equation 8.1 can be used to estimate the number of C•RAM clock cycles needed for a fault simulation round. Our goal is to find a fault-to-pattern ratio that minimizes the cost of the next two round.

When determining the next fault-to-pattern ratio, only three scenarios are considered, namely, doubling p_{prev} (the previous pattern parallelization factor), keeping p_{prev} unchanged, or halving p_{prev}. It is necessary to determine the total cost to fault simulate the next $2p_{prev}$ test patterns in all three scenarios. The cost for running one, two and four rounds are thus calculated.

Note that there are two more variables in Equation 8.1. These variables, $n_{undetected}$ and $n_{triggered}$, are not known in most of the cases. In order to estimate these values for evaluating the round cost estimate, it is assumed that the *fault dropping rate*, d, and *fault-group triggering ratio*, t, are going to be similar for the next $2p_{prev}$ test patterns. Thus

$$d = \frac{D_0}{D_1} = \frac{D_1}{D_2} = \ldots \text{ and } t = \frac{T_1}{D_1} = \frac{T_2}{D_2} = \ldots \tag{8.4}$$

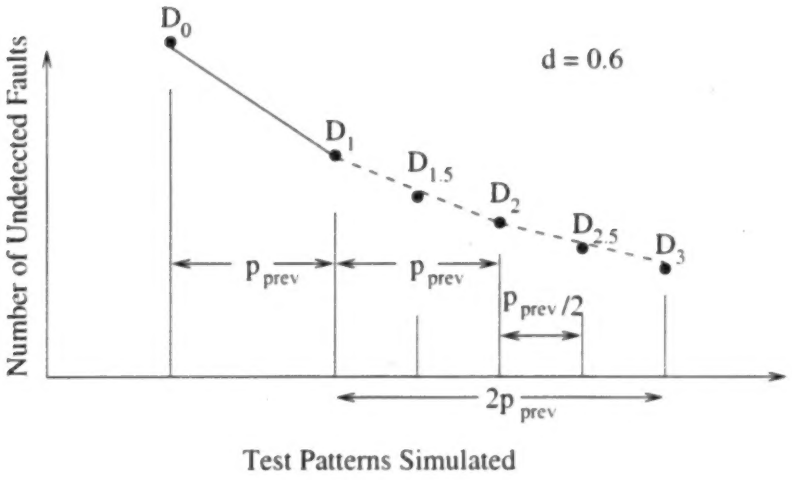


Figure 8.8: Choosing the Best Fault-to-Pattern Ratio

where D_1 is the current number of undetected faults after the last round. D_0 is that number before the last round (the last p_{prev} test patterns), and T_1 is the number of fault-groups triggered in the last round. Using Equation 8.4, the values of $D_{1.5}$, D_2, and others shown in Figure 8.8 can be estimated. These values are then used to calculate the total round cost for the three alternative future scenarios. If p_{prev} is already at a predetermined maximum (minimum) value, then doubling (halving) p_{prev} is not considered. The scenario that implies the minimum cost is then chosen as the fault-to-pattern ratio for the next round.

$$T_{doubling} = GST + CMT * D_1 + \frac{FST * T_1 * 2 * p_{prev}}{\#PEs} \tag{8.5}$$

$$T_{current} = 2 * GST + CMT * (D_1 + D_2) + \frac{FST * (T_1 + T_2) * p_{prev}}{\#PEs} \tag{8.6}$$

$$T_{halving} = 4*GST+CMT*(D_1+D_{1.5}+D_2+D_{2.5})+\frac{FST * (T_1 + T_{1.5} + T_2 + T_{2.5}) * \frac{p_{prev}}{2}}{\#PEs} \tag{8.7}$$

For the doubling p_{prev} scenario, Equation 8.5 is used. In this scenario, p is doubled and only one round is needed to simulate the next $2p_{prev}$ test patterns. The cost can thus be estimated with D_1 and T_1. In a scenario where we keep p unchanged, two rounds will be needed to simulate the next $2p_{prev}$ test patterns. In Figure 8.8, we can see that the first round will simulate from D_1 to D_2, and the next round carries

Circuit	Fault Simulation Time in Seconds and Speed-Up			
	simf_pps	simf_fps	static hps	dynamic hps
c1355nr	0.190s (3.37)	1.756s (0.36)	0.108s (5.93)	0.108s (**5.94**)
c1908nr	0.259s (5.77)	12.615s (0.12)	0.221s (6.78)	0.196s (**7.62**)
c2670nr	6.675s (**10.19**)	1365.713s (0.05)	11.708s (5.81)	7.256s (9.37)
c3540nr	0.358s (**4.71**)	12.956s (0.13)	0.390s (4.33)	0.439s (3.85)
c5315nr	0.443s (**4.63**)	14.415s (0.14)	0.466s (4.40)	0.459s (4.47)
c6288nr	1.730s (5.46)	3.141s (3.01)	0.525s (18.00)	0.517s (**18.26**)
c7552nr	9.170s (8.38)	3971.439s (0.02)	20.636s (3.72)	8.850s (**8.68**)

Table 8.3: Final Results

on from D_2 up to D_3. The cost of this scenario is thus the sum of the cost of these two rounds. D_1, T_1, and $p = p_{prev}$ is substituted into the Equation 8.1 to calculate the cost of the first round. If the fault dropping rate and the fault triggering ratio are unchanged (which is our assumption), the second round will start at D_2 and T_2. These values are again substituted into Equation 8.1 to estimate the cost of the second round. Adding these two cost functions, we obtain Equation 8.6. Halving p_{prev} will result in four rounds, and the cost is estimated similarly. Equation 8.7 is the resulting cost function for setting $p = p_{prev}/2$.

8.3 Evaluation

As mentioned in the previous section, most of the run-time is spent in good simulation, creating fault simulation masks ($CreateMask$), and fault simulation. These are the major activities in our fault simulation algorithm. In dynamic-hybrid fault simulation, additional time is needed to perform the ratio calculations and to create new partition masks.

Table 8.3 lists the performance of all the simulators evaluated in this thesis. The best result for each circuit is highlighted with bold-type font. The non-dfs version of *simf_fps*[2] is used for obtaining the fault-parallel fault simulation results. On the other hand, both the static and the dynamic simulators have depth-first-search enabled and the C•RAM mapper disabled. Four partitions with 256 PEs each were used for

[2]The version of *simf_fps* that do not use the Depth-First-Search-from-primary-output heuristic.

the static-hybrid fault simulator, while 16 partitions (64 PEs per partitions) were used initially in the dynamic one. These partition sizes were selected after some experimentation.

From the discussion in Chapter 4, the hybrid fault simulators would be expected to run faster than all the other ones. However, this is not always the case according to the results in Table 8.3. This can be explained by the following reasons.

The theory discussed in Chapter 4 did not take into account some of the overhead associated with decreasing p. This overhead includes the good simulation run time and the fault triggering checking time. From Equations 8.5, 8.6, and 8.7, we should remember that as p decreases, the GST component also increases (halving p implies doubling the coefficient of GST). This means increased fault simulation overhead.

There are also problems on the technical side. More time is spent in fault checking not only because of the smaller p, but also because more time is needed to check the primary effects of faults in the fault-groups. As a result, the overhead per round is generally higher for the hybrid-parallel fault simulators than for the pattern-parallel one. It may be worthwhile to develop a hybrid fault simulator that can speed up or even eliminate the checking of primary effects. This will require a whole new design.

Another technical problem is that when faults are simulated in parallel, the early stopping technique used in the pattern-parallel fault simulator cannot be applied to the hybrid ones. In the pattern-parallel fault simulator, when the fault effects propagate to at least one primary output during fault simulation, the fault can be considered detected without further gate evaluations. This can reduce a lot of unnecessary gate evaluations. However, this tactic cannot be applied in a hybrid fault simulator efficiently because the checking will have to make sure that a fault is detected in each of the partitions, which can be very time consuming considering that some partitions in the simulation may not detect the fault at all, thus wasting all the time spent in checking. This problem should only significantly affect the performance when detecting the faults early on in the simulation, where faults tend to be easier to detect. When the fault coverage is already very high, most of the remaining faults cannot be detected very easily anyway, thus the early stopping effect is not as significant.

Figure 8.9 illustrates the effect of running the static-hybrid fault simulator as the

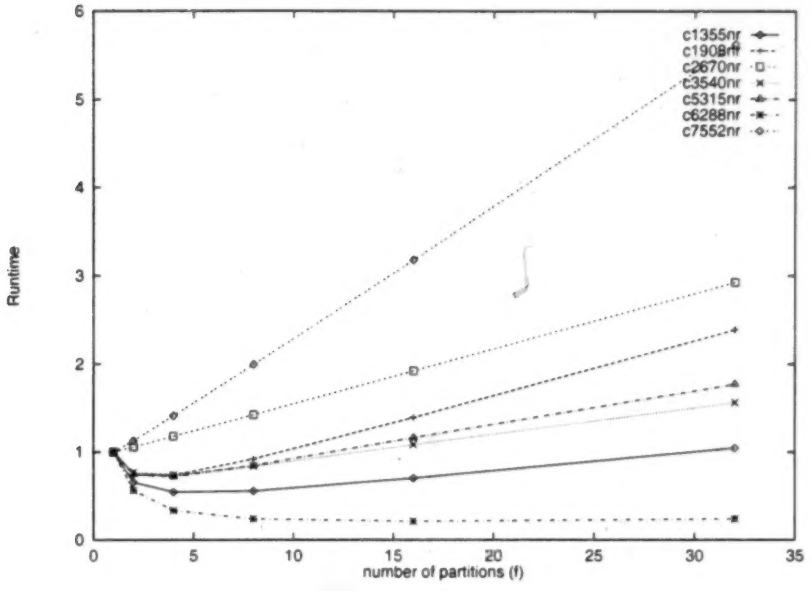


Figure 8.9: Effect of Using Different Number of Partitions (Normalized to f=1)

number of partitions is varied. The curve is normalized with respect to the run-time with only one partition. Again, the simulations are run up to 99% fault coverage. It can be seen that there are two extremes among the seven of the benchmark circuits. Note that the easier circuits tend to run faster with more partitions (e.g. c6288nr). On the other hand, the harder circuits such as c2670nr and c7552nr run much faster with fewer partitions. The remaining four benchmark circuits run fastest with 4 partitions (each with 256 test patterns).

Ideally, the dynamic-hybrid fault simulator should recognize the changes in performance and adjust the fault-to-pattern ratio accordingly. However, the current implementation of our dynamic-hybrid fault simulator is inadequate for doing so. Let's take c6288nr for example. Initially, we run the simulation with 16 partitions, assuming 1024 PEs. Since the c6288nr circuit runs best with more partitions (smaller p), an ideal dynamic-hybrid fault simulator should increase the number of partitions after each round (halving p). In our experiment, however, this is not happening. In fact, our simulator has never reached the point where the p halving option won over the other two options (with 1024 PEs). This fact can be interpreted in two ways. (1) We have never reached such occasion or (2) the function is inadequate.

In the three cost equations, the only term that has an exact value is *Good Simulation Time*. The other two terms are only approximations. In this design, it was assumed that the fault dropping rate and the fault triggering ratio are going to be unchanged throughout the next $2p_{prev}$ test patterns. This assumption may not be accurate enough for the relatively small benchmark circuits that we considered. Also, the terms CMT and FST are approximations from the structure of the circuit and statistical data from the previous round. We could possibly improve the dynamic-hybrid fault simulator by developing a better function for selecting the next fault-to-pattern ratios. Currently, the initial number of partitions is chosen by trial and error. It would be desirable to develop a better method to find the initial ratio as a result of an initial heuristic evaluation of the circuit.

8.3.1 PE Utilization

PE utilization was used to evaluate *simf_pps* and *simf_fps*. We would want to find out PE utilization for *simf_hps* also so that we can compare the fault simulators. Since the hybrid fault simulators use the fault group data structure, special techniques must be used to find out the PE utilization ratio.

Recall that the PE utilization equals the number of PEs doing useful fault simulations divided by the number of PEs used for fault simulations. The denominator is simply the number of passes times the number of PEs. In order to measure the numerator, we must first define the idea of a *useful PE* in the presence of fault groups. When a fault group is fault simulated in a partition, only the set of test patterns that could trigger at least one of the faults in the fault group can be considered useful. For each of the undetected faults in the fault group, if it is not detected by the current set of test patterns, then all the test patterns that trigger this fault are useful. On the other hand, if the fault is detected, then only the fault triggering test patterns in the partition up to and including the pattern that detects the fault are useful. The set of useful PEs (test patterns) is thus the union of these individual useful sets. If none of the faults were detected, then the useful set of PEs equals the set of PEs that triggers the faults. The total number of useful PEs is thus the accumulation of the number of PEs in the each useful set.

Figure 8.10 illustrates how a useful set of PEs is determined. Let's consider a fault

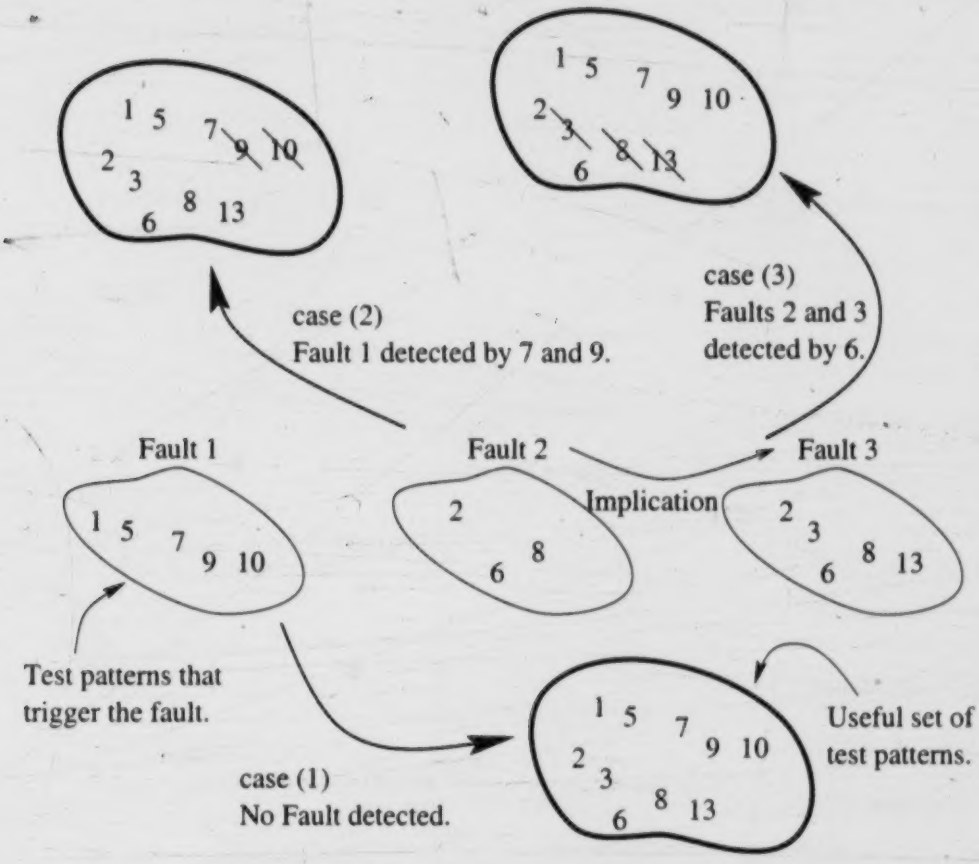


Figure 8.10: Determining a Useful Set of PEs

group with the three faults 1, 2 and 3 with at least 13 PEs. Each fault is triggered by a set of test patterns (running on different PEs). In case (1), none of these faults is detected by any of these test patterns. The useful set of PEs (in bold lines) in this case is thus simply the union of all the sets of PEs. In case (2), fault 1 is detected by two test patterns 7 and 9. Since the PE 7 is sufficient to detect the fault if we simulate the patterns with less parallelism, PE 9 is not a useful PE even though it can also detect the fault. As a result, only the PEs triggering fault 1 before and including PE 7, plus the ones triggering the other faults, are useful. In case (3), fault 2 is detected at PE 6. The useful PEs are thus 2 and 6 plus the ones that triggered fault 1. Although PE 3 also triggers fault 3, it is not useful because of the fault implication effect. When we determine that fault 2 is detected, we immediately know that fault 3

Circuit	Dynamic-Hybrid		Static-Hybrid		*simf_pps*	*simf_fps*
	needed/used	util.	needed/used	util.	util.	util.
c1355nr	9k/285k	3.14%	10k/230k	**4.51%**	1.54%	1.55%
c1908nr	54k/654k	8.19%	55k/598k	**9.18%**	6.03%	1.96%
c2670nr	17m/50m	**34.21%**	17m/50m	34.15%	22.17%	6.83%
c3540nr	92k/1083k	8.50%	103k/975k	**10.52%**	3.65%	7.77%
c5315nr	119k/1184k	10.01%	136k/1108k	12.28%	2.56%	**14.98%**
c6288nr	11k/624k	1.69%	14k/231k	6.99%	0.23%	**8.52%**
c7552nr	7208k/27m	**27.11%**	7267k/27m	26.70%	19.28%	2.82%

Table 8.4: PE Utilizations of the Hybrid Fault Simulators

is also detected. In this case we do not need to find the triggering conditions of fault 3 at all, thus the extra PEs that triggers fault 3 cannot be considered useful.

The PE utilizations for the static-hybrid fault simulator and the dynamic-hybrid fault simulator are recorded in Table 8.4. The PE utilizations for *simf_pps* and *simf_fps* are also listed in the same table. It can be seen that PE utilizations for the hybrid fault simulators are generally higher than the non-hybrid ones. When compared to *simf_pps*, both hybrid fault simulators have higher PE utilizations than the pattern-parallel fault simulator for all circuits. This proves that our implementation of the hybrid fault simulators succeeded in reducing the number of wasted PEs in pattern-parallel fault simulation. When compared to *simf_fps*, we found that the hybrid fault simulators have a lower PE utilization only when the faults are very easy to detect (i.e. for circuits c5315nr and c6288nr). This is because only relatively few test patterns were fed into these circuits, and these are the best cases for the fault-parallel fault simulator.

Figure 8.11 shows the accumulated PE utilization ratios at different points during fault simulation of circuit c2670nr. The curve for fault-parallel fault simulation drops like a $1/x$ curve. This is because the best cases for fault-parallel fault simulation is at the beginning of the fault simulation, where there are many undetected faults such that each C•RAM PE is assigned a fault. As faults are detected and dropped from the fault list, the number of C•RAM PEs exceeds the number of undetected faults. This results in wasted PEs, which reduces the final PE utilization ratio. The curve for pattern-parallel fault simulator, on the other hand, starts with very low PE

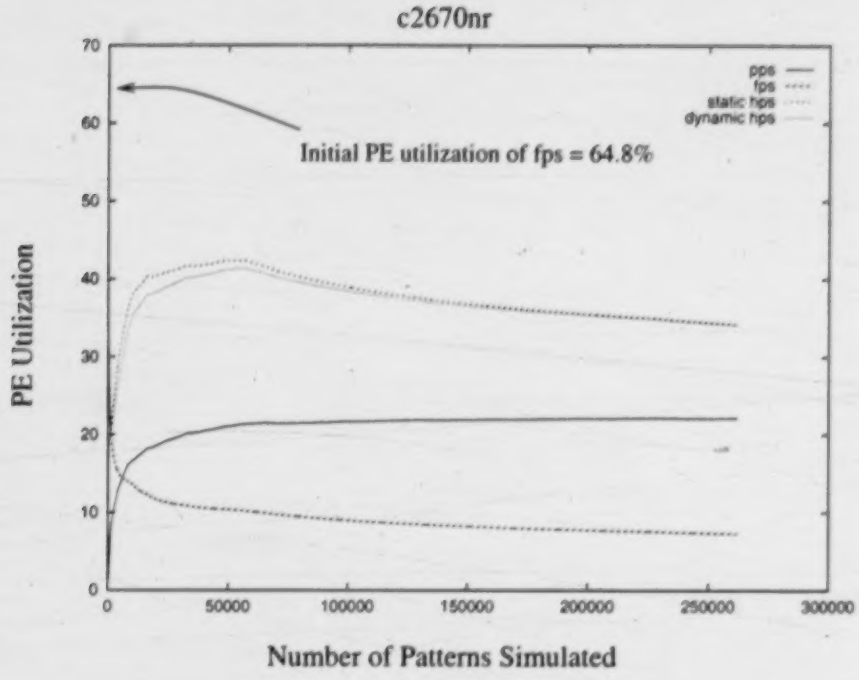


Figure 8.11: PE Utilization Curve

utilization. This is because there are many easy to detect faults at the beginning of the fault simulation. These faults can be detected by the very first test patterns that triggered them, thus utilizing only 1/1024 PEs. Eventually, the fault list contains only the faults that are relatively hard to detect. The test patterns may be capable of triggering the faults, but not detecting them. If these faults are triggered every $n << 1024$ test patterns, then the PE utilization ratio will be driven up. The PE utilization will not reach 100% as there is an upper bound $(1/n)$. If $n = 5$ then the upper bound is about 20%.

The curves for the two hybrid fault simulators are very similar. Both curves start with low PE utilizations. However, the ratio increases more rapidly than for the pattern-parallel fault simulator. This is because of the C•RAM partitioning, which reduces the number of PEs spent detecting "easy" faults. If four partitions are used, then the utilization is 1/256, which is 4 times greater than that of *simf_pps*. In our graph, as more patterns are simulated, the PE utilization drops. This is a non-trivial effect because it depends on the structure of the circuit and the test patterns. If the

circuit	Fault Simulation Time in seconds and Speed-Up			
	simf_pps	*simf_fps*	static hps	dynamic hps
c1355nr	0.189s (3.38)	2.567s (0.25)	0.115s (**5.59**)	0.124s (5.18)
c1908nr	0.256s (5.86)	14.937s (0.10)	0.220s (6.82)	0.193s (**7.76**)
c2670nr	4.574s (**14.87**)	1446.264s (0.05)	10.313s (6.60)	4.773s (14.25)
c3540nr	0.334s (5.06)	13.735s (0.12)	0.332s (**5.09**)	0.362s (4.67)
c5315nr	0.435s (**4.72**)	14.678s (0.14)	0.436s (4.70)	0.536s (3.82)
c6288nr	1.732s (5.45)	3.220s (2.93)	0.432s (**21.88**)	0.593s (15.94)
c7552nr	5.365s (14.32)	4027.996s (0.02)	19.888s (3.86)	5.283s (**14.54**)

Table 8.5: Fault Simulation Results with 4K PEs

Circuit	simf_pps	simf_fps	static hps	dynamic hps
c1355nr	1.0	0.9	**1.8**	1.7
c1908nr	1.0	1.3	**1.5**	**1.5**
c2670nr	1.3	1.3	1.3	**1.4**
c3540nr	1.1	1.2	1.9	**2.0**
c5315nr	1.0	1.0	**1.6**	1.5
c6288nr	1.0	1.1	**2.4**	1.6
c7552nr	1.3	1.3	1.1	**1.4**

Table 8.6: Speed-up of Fault Simulations Going from 1K PEs to 4K PEs

test patterns are capable of triggering the "hard" faults very frequently, then the PE utilization ratio will rise. For example, if the PE utilization ratio is 40% when there are only "hard" faults left, then the PE utilization ratio will drop if the faults are triggered every 3 test patterns or more (approaches $1/3 = 33\%$). However, if the triggering frequency is every 2 test patterns, then the PE utilization will rise (it will approach $1/2 = 50\%$).

8.3.2 Simulating Using a Larger C•RAM

Finally, we evaluate the C•RAM fault simulators by emulating a larger C•RAM. This evaluation can test the capacity of our fault simulators for a more realistic C•RAM platform. In our evaluation, a system with 4K PEs and 16K bits of local memory per PE is emulated. This equals a system with 8M bytes of RAM. For CAD workstations, this is actually a very small amount of memory. We chose this configuration only because it allows us to obtain results in a reasonable amount of emulation time.

Table 8.5 shows the performance of the four C•RAM fault simulators using 4K PEs. By increasing the number of PEs, we obtain more dramatic performance improvement over the simulator without C•RAM. In the table of final results we presented at the beginning of this section. *simf_pps* runs faster than the hybrid fault simulators in three benchmark circuits. We can see from Table 8.5 that this number is reduced to one.

The speed-up ratio from going from 1K PEs to 4K PEs is shown in Table 8.6. The speed-up ratio for the hybrid fault simulators is generally higher than that of the pattern-parallel fault simulator's. A higher ratio implies a more scalable solution. This suggests that the hybrid fault simulators are more scalable than the non-hybrid ones. In addition, the dynamic hybrid fault simulator is more scalable than the static hybrid fault simulator when simulating circuits that are "harder" (need more test patterns to reach 99% fault coverage). This is due to the adapting ability of the dynamic hybrid fault simulator which allows it to run in a purely pattern-parallel fashion when the fault dropping rate is low.

Figure 8.12 shows the speed-up versus the number of C•RAM PEs for four of our benchmark circuits[3]. The scalability of the four simulators can be observed clearly from these graphs. We can see that the scalability of the dynamic hybrid fault simulator is better than the other ones in three of the four circuits (c3540nr, c6288nr, and c7552nr). It is interesting to note that the dynamic hybrid fault simulator does not speed-up as much as the static hybrid fault simulator and *simf_pps* for circuit c2670nr. This situation can happen when the faults are not only hard to detect but also hard to trigger. Let's consider three faults with fault triggering rate of about 1/16K for example. On a C•RAM system with 16K of PEs running static hybrid fault simulation using 4 partitions, only one simulation pass is needed. On the other hand, if dynamic hybrid fault simulation with only one partition is used, then three simulation passes are needed. An ideal dynamic hybrid fault simulator should have analyzed the circuit structure and determined that four partitions should be used.

[3]There are three data points in each curve: 1K, 4K and 16K PEs. The first two data points were taken with 64 PEs per C•RAM partitions at the beginning of dynamic-hybrid fault simulation. In contrast, the data points with 16K PEs had 256 PEs per C•RAM partitions. Since the configurations were different, we may not compare the 16K PE results with the other results directly. Therefore, the simulation results with 4K PEs were used in the previous paragraphs to represent a larger C•RAM system.

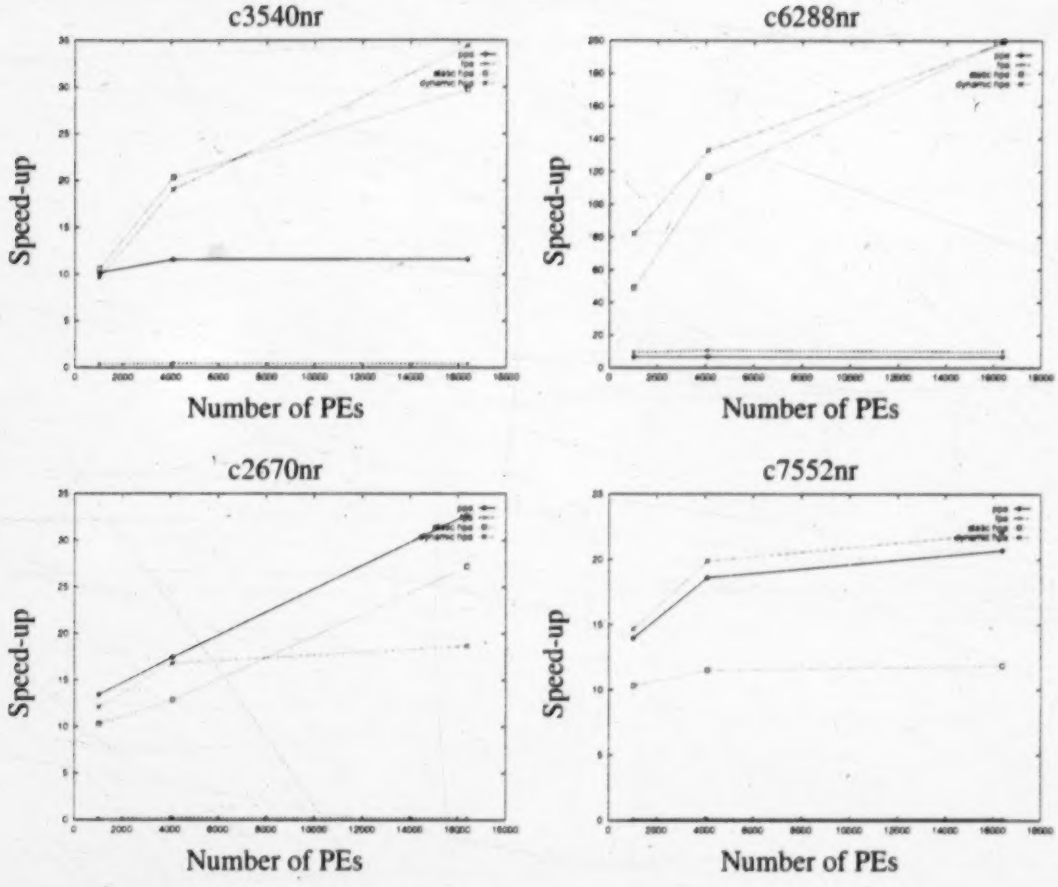


Figure 8.12: Speed-Up Graphs for c2670nr, c3540nr, c6288nr, and c7552nr

The pattern-parallel fault simulator can also run faster than the dynamic hybrid fault simulator in this situation because it requires less overhead. One way to fix the dynamic hybrid fault simulator is to use the optimized pattern-parallel fault simulation code as soon as it falls back to only one partition.

Chapter 9

Conclusion

9.1 Summary of Results

A benchmark conventional fault simulator *simf* was developed initially to gain experience with implementing the basic algorithms and to establish a point of comparison for other fault simulators. It was shown to be efficient and typically faster than the public domain fault simulator *sim3*. Three different experimental C•RAM fault simulators were designed, namely *simf_pps*, *simf_fps* and *simf_hps*. In addition, two different versions of *simf_hps* were designed. One implements a static-hybrid fault simulation scheme while the other one implements a dynamic-hybrid fault simulation scheme. All the source code is included in the Appendix. The various optimizations used in each fault simulator is summarized in Table 9.1.

Simf_pps is a pattern-parallel fault simulator which is based on the famous PPSFP algorithm (Parallel Pattern Single Fault Propagation). It was found to be highly efficient and relatively easy to implement on C•RAM. However, this solution may not be scalable as the size of C•RAM increases because of the low PE utilization at the beginning of the simulation, where most of the faults are relatively easy to detect.

Simf_fps implements a fault-parallel fault simulation algorithm. This simulator runs our benchmark problems even more slowly than *simf* most of the time. This is primarily due to two bottlenecks: single pattern fault-free simulation and fault triggering condition checking. However, with benchmark circuit c6288nr, *simf_fps* is even faster than *simf_pps*. This supports our claim that fault-parallel fault simulation tends to be more efficient when there are many easy-to-detect faults. Techniques developed when designing this simulator were re-used in *simf_hps*.

Optimization	simf	PPS	FPS	HPS
Event Heap	yes	yes	yes	yes
Fault Implication	yes	yes	no	yes
DFS Fault Grouping	no	no	no	yes
Fault-Family	no	no	yes	yes
Node Allocation*	no	no	yes	no
Fault Triggering Test	yes	yes	yes	yes
Early Stopping	yes	yes	no	no
Simulating from Site of Fault(s)	yes	yes	yes**	yes**

* use of the cram_mapper module

** Multiple faults were inserted

Table 9.1: Optimization Implemented in Each Fault Simulator

Simf_hps is a hybrid fault simulator that parallelizes along both the fault and the pattern dimensions. A *static* version was first developed, where the fault-to-pattern ratio is fixed. This simulator was found to be efficient; however, it was not as fast as *simf_pps* for four of the seven benchmark circuits when 1024 PEs were used. Extra functions were then added to this simulator to make it *dynamic*, where the fault-to-pattern ratio is adjustable. The dynamic version can sometimes provide better results than the static-hybrid fault simulator. However, compared to *simf_pps*, the two hybrid fault simulators are not consistently faster. Also, for the benchmark circuits at least, the dynamic hybrid method was not always faster than the static hybrid method.

In this thesis, a software emulation of C•RAM with 1024 PEs was used to run the algorithms. Tests with 4096 PEs were also run to investigate the scalability of the C•RAM fault simulators. It was found that the hybrid fault simulators are more scalable than the non-hybrid ones. Since 4096 PEs would be found in 8M bytes of C•RAM, this could be considered a small system. It would be desirable to emulate an even larger C•RAM because that would be closer to real memory capacities found on today's workstations. However, this was not done because C•RAM emulation is very time consuming. Nevertheless, with respect to a host without C•RAM, a host and a C•RAM with 1024 PEs was estimated to be capable of accelerating fault simulation by 10 times using appropriate algorithms, and a host and a C•RAM with 4096 PEs can accelerate fault simulation by about 10 to 30 times. The notion of PE utilization

circuit	Fault Simulation Time in seconds and Speed-Up			
	simf_pps	*simf_fps*	static hps	dynamic hps
c1355nr	0.069s (**9.24**)	2.534s (0.25)	0.094s (6.78)	0.099s (6.44)
c1908nr	0.105s (**14.20**)	14.816s (0.10)	0.168s (8.90)	0.152s (9.87)
c2670nr	0.657s (**103.55**)	1427.177s (0.05)	5.014s (13.57)	0.717s (94.83)
c3540nr	0.194s (**8.71**)	13.534s (0.12)	0.252s (6.70)	0.276s (6.11)
c5315nr	0.294s (**6.98**)	14.415s (0.14)	0.357s (5.75)	0.431s (4.75)
c6288nr	0.392s (24.10)	3.142s (3.01)	0.355s (**26.63**)	0.525s (18.00)
c7552nr	1.463s (**52.52**)	4019.670s (0.02)	13.580s (5.66)	1.632s (47.09)

Table 9.2: Fault Simulation Results With 4K PEs Running at 143MHz

was introduced to evaluate the effectiveness of the algorithms in using C•RAM. It is found that PE utilization for the hybrid fault simulators can be up to two times that of the pattern-parallel fault simulator, depending on the circuit and the test patterns.

At the end of Chapter 4, we mentioned that the cycle time of the workstation is 17 times faster than the assumed C•RAM system. Under this assumption, our fault simulators can improve C•RAM up to 22 times (Figure 8.5). We assumed that our C•RAM system is based on the DRAM technology. Faster memory architecture technologies exist that can be used to implement C•RAM. Assuming C•RAM is implemented on a technology (e.g. static RAM) that runs as fast as the CPU (167MHz in our case), the speed-up of our algorithm, using 4K of PEs, is up to 103 times (see Table 9.2).

Note that in Table 9.2, the pattern-parallel fault simulator runs faster than the other ones in six out of the seven tests. This is because this algorithm uses the least CPU time. As the C•RAM speed improves, the CPU time becomes the main bottleneck. This indicates that the future research priority should be changed to find algorithms that use the least CPU time when the C•RAM is relatively fast.

9.2 Further Research

In order to apply C•RAM to fault-simulate industrial circuits, there are at least three remaining concerns. The primary concern is that most industrial circuits are much larger than our benchmark circuits. Second, most circuits in industry are sequential

circuits that include clocked memory elements as well as combinational logic. Finally, there are other popular fault models besides the ones used in this research. We will discuss each of these concerns in the following subsections.

9.2.1 Fault-Simulating Large Circuits

The fault simulators developed in this research are capable of simulating circuits as large as C•RAM can support. Up to now, C•RAM with 16K bits of local memory has been simulated. This means that the simulators can simulate circuits with up to roughly 12k gates. This is rather small compared to most industrial circuits.

One solution is to use a larger C•RAM. To be flexible, however, we may want to develop a fault simulator that does not depend on the upper limit of the amount of local memory per PE. This can be done by grouping multiple PE's together and to pool their local memory. For example, when two PEs are grouped together, we can have 32K bits of memory with one PE, at the cost of reducing the parallelism by a factor of two (the second PE is used as a memory "server"). This strategy would suffer the cost of occasionally having to transfer data among local memories by left and right shifting. Shifting data among PEs, however, is a relatively fast operation compared to a memory access. The overhead is hard to predict without simulation.

9.2.2 Sequential Circuits

We may want to expand the fault simulators to simulate sequential instead of combinational circuits, since most industrial circuits are sequential. We could upgrade the simulators developed in this research to simulate sequential circuits with the following modifications.

First, the fault simulator right now only simulates six types of gates (not even XOR gate). In order to simulate sequential circuits, we must be able to simulate D flip-flops. This involves defining fault-collapsing rules and fault-triggering rules for D flip-flops. In order to define these rules for the D flip-flops (also for XOR gate), one must decide on whether the circuit element should have a behavioural view or a structural view[1]. In a *behavioural view*, we only need to take care of the faults at the input and output of the gate. If we view the circuit element as a block of basic

[1]The two terms are taken from the VHDL hardware description language.

gates (*structural view*), faults at the internal wires should also be considered. The AND gates, OR gates and BUFFERs in this thesis used a structural view. Using a behavioural model for the flip-flops would allow the simple use of parallel `cint` variables to hold each bit of memory in the circuit.

Second, we must take care of the gate evaluation order. Currently, the heap sorts according to the gate number because this number reflects the gate evaluation order. When simulating sequential circuits, we will have to come up with a scheme to sort the gates so that the gate evaluation order is correct. We may sort the gates and use the gate number in a synchronous circuit. In asynchronous circuits, timing is very important. For example, if a signal propagating to the input of a D flip-flop arrives after the edge of the clock signal, the flip-flop will not latch in the correct value. In order to simulate asynchronous circuits, we must use actual propagation delays in gates, maybe even separate wires. The event heap should then use the signal timing information for sorting. There is also more trouble in performing fault-free simulations. Clearly a synchronous sequential fault simulator would be much easier to implement than an asynchronous sequential one.

9.2.3 Other Fault Models

In this research, only stuck-at, gate delay, and the transistor stuck-open models are considered. In industry, there is considerable on-going interest in the harder problem of *path-delay fault simulation*. The path-delay fault model assumes that signal delays are caused by accumulated small delays on the signal propagation path. This contrasts with the gate delay fault, where the spurious delay is modeled as being lumped at the output of one gate. Fault simulation of this fault model will require more sophisticated designs such as path generation and 6-value logic simulation. Reference [9] describes a method to fault-simulate path-delay faults by parallel processing of patterns. It would be a worthwhile and likely very challenging research project to expand the C•RAM fault simulator to simulate path-delay faults.

Bibliography

[1] Y. Aimoto, T. Kimura, Y. Yabe, H. Hejuchi, Y. Nakazawa, M. Motomura, T. Koga, Y. Fujita, M. Hamada, T. Tanigawa, H. Nobusawa, and K. Koyama. A 7.68GIPS 3.84 GB/s 1W Parallel Image-Processing RAM Integrating a 16Mb DRAM and 128 Processors. In *1996 IEEE International Solid-State Circuits Conference*, pages 372–373, 1996.

[2] T. Blank. The MasPas MP-1 Architecture. In *Proceedings of the IEEE Compcon Spring 1990*, pages 20–24. IEEE, February 1990.

[3] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combination Benchmark Circuits and a Target Translator in FORTRAN. In *1985 International Symposium on Circuits and Systems*, pages 663–698, June 1985.

[4] K. Cattell and J.C. Muzio. Synthesis of One-Dimenstional Linear Hybrid Cellular Automata. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, (3):325–335, March 1996.

[5] J. Childers and P. Reinecke. SVP: Serial Video Processor. In *Proceedings of IEEE 1990 Custom Integrated Circuits Conference*, pages 17.3.1–17.3.4, 1990.

[6] B.F. Cockburn and A. L.-C. Kwong. Transition Maximization Techniques for Enhancing the Two-Pattern Fault Coverage of Pseudorandom Test Pattern Generators. In *1998 IEEE VLSI Test Symposium*, pages 430–437, April 1998.

[7] D. G. Elliott. *Computational RAM: A Memory - SIMD Hybrid.* PhD thesis, University of Toronto, Dept. of Electrical Engineering, 1998.

[8] D. G. Elliott, W. M. Snelgrove, and M. Stumm. Computational RAM: A Memory-SIMD Hybrid and its Application to DSP. In *IEEE 1992 Custom Integrated Circuits Conference*, pages 30.6.1–30.6.4, May 1992.

[9] F. Fink, K. Fuchs, and M. H. Schulz. Robust and Nonrobust Path Delay Fault Simulation by Parallel Processing of Patterns. *IEEE Transactions on Computers*, 41(12):1527–1536, December 1992.

[10] M. J. Flynn. Very High-Speed Computing Systems. In *Proceedings of IEEE 54:12*, pages 1901–1909, December 1966.

[11] M. J. Folk and B. Zoellick. *File Structures*, chapter Cosequential Processing and the Sorting of the Larger Files, pages 280–284. Addison-Wesley, Reading, Mass., 1992.

[12] M. Gokhale, B. Holmes, and K. Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer*, pages 23–31, April 1995.

[13] D. Harel and B. Krishnamurthy. Is There Hope for Linear Time Fault Simulation? In *Proceedings of FTCS 17*, pages 28–33, 1987.

[14] J. P. Hayes. *Rational Fault Analysis*, chapter Modeling Faults in Digital Logic Circuits, pages 78–95. R. Saeks and S. R. Liberty, eds., Marcel Dekker, New York, 1977.

[15] R. A. Heaton and D. W. Blevins. BLITZEN: A VLSI Array Processing Chip. In *Proceedings of IEEE 1989 Custom Integrated Circuits Conference*, pages 12.1.1–12.1.5, 1989.

[16] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design*, chapter Fallacies and Pitfalls, pages 75–101. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1992.

[17] J. L. A. Hughes and E. J. McCluskey. An Analysis of the Multiple Fault Detection Capabilities of Single Stuck-at Fault Test Sets. In *Proceedings of International Test Conference*, pages 52–59, October 1984.

[18] N. Ishiura and S. Yajima. Dynamic Two-Dimensional Parallel Simulation Technique for High-Speed Fault Simulation on a Vector Processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(8):868–875, August 1990.

[19] M. Kurokawa, A. Hashiguchi, K. Nakamura, H. Okuda, K. Aoyama, T. Yamazaki, M. Ohki, M. Soneda, K. Seno, I. Kumata, M. Aikawa, H. Hanaki, and S. Iwase. 5.4GOPS Linear Array Architecture DSP for Video-Format Conversion. *1996 IEEE International Solid-State Circuits Conference*, pages 254–255, 1996.

[20] T. M. Le, S. Panchanathan, and M. Snelgrove. Computational-RAM Implementation of Mean-Average Scaleable Vector Quantization for Real-Time Progressive Image Transmission. *CCECE'96*, pages 442–445, 1996.

[21] J. LeBlanc. LOCST: A Built-In Self-Test Technique. *IEEE Design and Test of Computers*, 1(4):46–66, November 1984.

[22] M. Maresca and T. J. Fountain. Scanning the Issue, Massively Parallel Computers. *Proceedings of the IEEE*, 79(4):395–402, April 1991.

[23] V. Narayanan and V. Pitchumani. Fault Simulation on Massively Parallel SIMD Machines: Algorithm, Implementations and Results. *Journal of Electronic Testing: Theory and Applications*, 3(1):79–92, February 1992.

[24] T. M. Niermann, W.-T. Cheng, and J. H. Patel. PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator. *IEEE Transactions on Computer-Aided Design*, 11(2):198–207, February 1992.

[25] B. Parhami. SIMD Machines: Do They Have a Significant Future. *Computer Architecture News*, 23(4):19–22, September 1995.

[26] D. Parkinson, D. J. Hunt, and K. S. MacQueen. The AMT DAP 500. In *Digest of Papers: COMPCON Spring 88*, pages 196–199. IEEE Comput. Soc. Press, March 1988.

[27] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, pages 34–44, March and April 1997.

[28] S. Seshu. On an Improved Diagnosis Program. *IEE Trans. on Electronic Computers*, EC-12(2):76–79, February 1965.

[29] Thinking Machines Corporation, Cambridge, Mass., Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*, 6 edition, 1990.

[30] E. G. Ulrich and T. G. Baker. Concurrent Simulation of Nearly Identical Digital Networks. *Computer*, 7(4):39–44, April 1974.

[31] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy. Fault Simulation for Structured VLSI. *VLSI Systems Design*, 6(12):20–32, December 1985.

[32] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar. Transition Fault Simulation. *IEEE Design and Test of Computers*, 4(2):32–38, 1986.

[33] N. Yamashita, T. Kimura, Y. Fujita, Y. Aimoto, T. Manabe, S. Okazaki, K. Nakamura, and M. Yamashina. A 3.84 GIPS Integrated Memory Array Processor with 64 Processing Elements and a 2-Mb SRAM. *IEEE Journal of Solid-State Circuits*, 29(11):1336–1343, November 1994.

[34] S. Zhang. BIST Generators for Faults with Sequential Behavior. Master's thesis, University of Victoria, Dept. of Computer Science, May 1993.

Appendix A

Source Code

A.1 Basic Modules

A.1.1 structure.h

```
/*
 *
 * . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 *
 * Parallel Fault Simulators on the C*RAM Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L -C. Kwong, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 *
 * structure.h
 * this file collects the data structures used in the simulators.
 */

#ifndef INT_SIZE
#define   INT_SIZE  (sizeof(int )*8)
#endif

#define   NUM_PE     perfParams.processors

// naming faults in the fault_list
#define a_NO(a)   fault_list[6*a + 0]
#define a_PO(a)   fault_list[6*a + 1]
#define a_SF(a)   fault_list[6*a + 2]
#define a_SR(a)   fault_list[6*a + 3]
#define a_SA1(a)  fault_list[6*a + 4]
#define a_SA0(a)  fault_list[6*a + 5]

typedef enum {g_INPT, g_BUFF, g_NOT, g_AND, g_NAND,
              g_OR, g_NOR, g_XOR, g_XNOR, g_DUM} GateT;

            /* 0     1     2     3     4     5     6     7   */
typedef enum {f_SA0, f_SA1, f_SR, f_SF, f_NO, f_PO, f_iNO, f_iPO}
  FaultT;

/* NI = not exist */



typedef struct Wire_struct
{
    char     exist;              /* does this wire exist? */
    int      from_gate;          /* from which gate */
    int      to_gate;            /* to which gate */
                                 /* these are the index of gate,
                                    not the gate number */
    int      node_num;
} Wire;

typedef struct Gate_struct
{
    int    gate_num;
    GateT  gate_type; /* type of gate */
    char   n_fan_in,
           n_fan_out;
    int    *fan_in;
    int    *fan_out;   /* if there are no fan_out, n_fan_out = 1
                          and fan_out contains the gate_num
                          if there is fan_out, n_fan_out = #fan_out
                          and fan_out contains the list of fan_out,
                          excluding gate_num */
    int    input_num; /* if it's an input, then this field stores
                         the input number */

    int    out_node;
    int    *in_node;
} Gate;


typedef struct Circuit_struct
{
    int  num_wire;       Wire *wire_list;
    int  num_gate;       Gate *gate_list;
    int  num_input;      int  *input_list;
    int  num_output;     int  *output_list;
} Circuit;


// for use with the flag variable
#define fd_UNDETECTED     0
#define fd_DETECTED       1
#define fd_IMPLIED        2
#define fd_EQUIVAL        4
#define fd_NONEXIST       8
#define fd_DEBUG         16
#define fd_TRIGGERED     32

typedef struct Fault_struct Fault;
struct Fault_struct
{
    Fault      *next;        // point to next fault
    Fault      *implied1;    // first implied fault
    Fault      *implied2;    // second implied fault

    char       flag;
    FaultT     type;
    int        wire_num;

    int        gate_num;
    char       family;
};


typedef struct FaultFamily_struct FaultFamily;
struct FaultFamily_struct
{
    int    gate_num;    // Fault Family number this struct represents
    FaultT type;        // type of effect this fault family produces,
                        // which is either f_SA0 or f_SA1
    Fault *Member;      // link list of members in Fault Family

    FaultFamily *next;  // next fault family;
    FaultFamily *prev;  // prev fault family;

    FaultFamily *nextinsim; // the list of fault family in simulation
};

typedef struct FaultGroup_struct FaultGroup;

#include <cram.h>

struct FaultGroup_struct
{
    int gate_num;
    FaultFamily *FF[2];

    Fault    *TrigFault[2];   // -> the first member in FF triggered
    cint     *TrigFlag[2];    // -> a mask of the triggered fault

    FaultGroup *dfs_next;
    FaultGroup *dfs_prev;
};

// end of structure.h
```

A.1.2 prototype.h

```
/*

 *
 * . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 *
 * Parallel Fault Simulators on the CoWS Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 *
 * prototype.h
 * this file contains function prototypes for proper linking.
 */

/* read_iscas.C */
extern void Read_ISCAS(char *filename, Circuit *cut);
extern void Print_Circuit(Circuit cut, int  level);
extern int  CacheEval (Circuit *cut);


/* gen_fault.C */
extern void Gen_Fault(Circuit *cut, Fault *fault_list, char sim_fault);
extern int  Restruct_Fault_List(Fault **fault_head, Fault *fault_list);
extern FaultFamily *Gen_Fault_Family (Circuit *cut, Fault *fault_list);
extern void Print_Fault_List(Fault *fault_list, Fault *fault_head);
extern void Get_Count(int  *n_sa, int  *n_d, int  *n_so);
extern void Count_Fault_Head(Fault *fault_head,
                             int *n_sa, int *n_d, int *n_so);
extern void Count_Fault_List(Fault *fault_list,
                             int *n_sa, int *n_d, int *n_so);
extern void Print_Fault(Fault *fault_list, int  level);
extern void DrawNode (Fault *fault_list, int level);
extern char *fault_table[8];
extern char *gate_table[10];
extern int  g_total_sa, g_total_d, g_total_so;


/* simulate.C */
extern int  Fault_Sim(Circuit *cut, Fault *fault_list, int  n_vector);


/* fault_family.C */
extern void InitFamily (Circuit *cut);
extern void AddToFamily (Fault *fault, int gate_num, int type);
extern void TransferFamily (int from_gate_num, int from_type,
                            int to_gate_num, int to_type);
extern FaultFamily *GetFaultFamily();
extern void PrintFaultFamily ();
extern FaultGroup *InitFaultGroup(Circuit *cut, Fault *fault_list);
extern bool CleanFaultGroup(FaultGroup *fg);
extern int  GetNumFaultGroup();


// end of prototype.h
```

A.1.3 simf.C

```
/*

 *
 * . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 *
 * Parallel Fault Simulators on the CoWS Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
 *
 * simf.C
 *
 * Purpose:  This is the main body of a fault simulator, simf.
 *           The simulator simulates Stuck-At, Stuck-Open, and
 *           Gate-Delay faults.
 *
 * Command Line Options:
 *   See the Usage() routine.
 */

#include <stdio.h>
#include <stdlib.h>
#include "structure.h"
#include "prototype.h"
#include "stopwatch.h"

void Usage()
{
  printf ("Usage: \n\
    -f name  : specifies circuit file\n\
    -l n     : number_of_vector\n\
    -v       : verbose mode (level : 1 to 2)\n\
    -s       : simulate stuck-at,gate-delay and stuck-open fault\n\
    -p       : Number of patterns per partition (default=256)\n\n");
  exit (1);
}

void Copyright()
{
  printf ("\
\nCopyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, AB, Canada \
\nAll rights reserved. \
");
}

int  g_total_sa, g_total_d, g_total_so;
int  NumPV = 256;
int  NumPartition;

int main (int argc, char *argv[])
{
  Copyright();

  char   *circuit_file=NULL,  /* name of circuit file  */
  int    n_vector=0,       /* number of input vectors */
  int    verbose=0,        /* detailed output" */

  char    sim_fault=0,      /* bit 0 = stuck-at */
                            /* bit 1 = stuck-open */
                            /* bit 2 = gate delay */

  /* read command line arguments */
MEASURECLOCK;
ZEROCLOCK;
STARTCLOCK;

  while (--argc)
  {
    if (argv[argc][0] == '-')
    {
      switch (argv[argc][1])
      {
        case 'f' : circuit_file = argv[argc+1]; break;
        case 'l' : n_vector = atoi (argv[argc+1]); break;
        case 'v' : verbose = atoi (argv[argc+1]);
                   if (!verbose || verbose > 3) verbose = 1;
                   printf ("verbose level = %d\n", verbose);
                   break;

        case 's' : sim_fault |= 1; break;   // stuck-at
        case 'p' : NumPV = atoi(argv[argc+1]); break;

        default  : Usage (); break;
      }
    }
  }
  if (perfParams.processors < NumPV)
  {
    fprintf (stderr,"Please set p < the number of PEs \n");
    exit (1);
  }
  NumPartition = perfParams.processors / NumPV;

  if (circuit_file == NULL) Usage();



  /* Read Circuit */
  Circuit   cut;            /* circuit data structure  */
  Read_ISCAS (circuit_file, &cut);
  //printf ("Circuit %s is read successfully\n", circuit_file);
  Print_Circuit (cut, verbose);


  /* Generate Fault list */
  Fault *fault_list;
  fault_list = (Fault *) malloc (sizeof (Fault) * (6 * cut.num_wire));
  if (!fault_list)
  {
    fprintf (stderr,"cannot allocate memory for fault_list\n");
    exit (1);
  }
  Gen_Fault (&cut, fault_list, sim_fault);

  Count_Fault_List(fault_list, &g_total_sa, &g_total_d, &g_total_so);



  /* Simulate */
  if (n_vector) Fault_Sim (&cut, fault_list, n_vector);

  /* Print  Results */
```

```
  int  new_ss, new_d, new_se;
  Count_Fault_List(fault_list, &new_ss, &new_d, &new_se);

  printf ("\
\n\
\n-------------------------------------------------\
\n  Fault Type  Total  Undetected  %%Coverage   \
\n-------------------------------------------------\
\n    Stuck-At   %6d    %6d     %6.2f\
\n     Delay     %6d    %6d     %6.2f\
\n  Stuck-Open   %6d    %6d     %6.2f\
\n-------------------------------------------------\n\n\n",
  g_total_ss, new_ss, (g_total_ss - new_ss)*100.00/g_total_ss,
  g_total_d , new_d , (g_total_d  - new_d )*100.00/g_total_d ,
  g_total_se, new_se, (g_total_se - new_se)*100.00/g_total_se);

  if (verbose) Print_Fault (fault_list, verbose);
  fflush (stdout);

STOPCLOCK;
PRINTCLOCK;

  printf ("\n");
  printf ("\n");
  printf ("Total time = %f + %f - %d + %f = %fs\n",
      cvar.time/1000000000.0,  STOPWATCH_TIME/1000.0,
        STOPWATCH_CALLED, STOPWATCH_OVERHEAD,
      cvar.time/1000000000.0 + STOPWATCH_TIME/1000.0
      - STOPWATCH_CALLED * STOPWATCH_OVERHEAD);
  printf ("\n");
  return 0;
}

// end of sim.C
```

A.1.4 read_iscas.C

```
/*

 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*RAN Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L -C. Hwang, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * read_iscas.c
 *
 * Purpose:  Read from an iscas netlist file and produce the circuit data
 *           described in structure.h file.
 *
 * Note:  iscas file format:
 *       - each line describes a piece of wire or a list of references
 *         wire:  1 1gat inpt 0 0 ...
 *          wirenumber
 *          wire type : gate output/fan out
 *          gate type : buff/not/and/nand/or/nor/xor/xnor/from
 *          # of fanout
 *          # of fanin
 *       - if fanout and fanin, then fanin first follow by fanout
 *       - fan out is represented by a wire with 1FY as wire type
 *       - fan in is a list of wire number
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "structure.h"
#include "prototype.h"

int  Replace_Gate (Circuit *cut);
int  read_line (FILE *fp);

#define BUF_SIZE 256
char *buffer;
int  buf_size = BUF_SIZE;

/*
 * Read_ISCAS
 *
 * Pre   : It is assume that cut is empty
 *     lines in the circuit file does not exceed 256 chars
 *
 * Input  filename = iscas file name
 *     options  = read options ( to be specified )
 *
 * Output  *cut = circuit data structure
 *
 * return: 1 successful, 0 unsuccessful
 *
 * Revision History
 * 97-06-07 removed all the unnecessary parts
 *      replaced all line/Line with wire/Wire
 *
 */

void Read_ISCAS (char *filename, Circuit *cut)
{
  #define INI_LIST_SIZE 1024
  FILE *circuit_fp;

  int  temp_num_wire, temp_num_gate, temp_num_input, temp_num_output;

  /* these temp variables hold the size of the list at the moment */
  temp_num_wire =
  temp_num_gate =
  temp_num_input =
  temp_num_output = INI_LIST_SIZE;

  /* init the Circuit Data structure */
  {
    cut->num_wire   = 0;
    cut->num_gate   = 0;
    cut->num_input  = 0;
    cut->num_output = 0;

    /* all lists are initialized to INI_LIST_SIZE, and will be
     shrink after the whole file is read */

    cut->wire_list  = (Wire*) malloc (sizeof(Wire) * temp_num_wire);
    cut->gate_list  = (Gate*) malloc (sizeof(Gate) * temp_num_gate);
    cut->input_list = (int *) malloc (sizeof(int) * temp_num_input);
    cut->output_list= (int *) malloc (sizeof(int) * temp_num_output);
    memset (cut->wire_list, 0, sizeof (Wire) * INI_LIST_SIZE);

    if (!(cut->wire_list && cut->gate_list && cut->input_list &&
        cut->output_list))
    {
      fprintf (stderr, "cannot allocate enough memory\n");
      exit (1);
    }
  }

  circuit_fp = fopen (filename, "r");
  if (!circuit_fp)
  {
    fprintf (stderr, "Cannot open %s\n", filename);
    exit (1);
  }

  /* read circuit */
  {

    buffer = (char *) malloc (sizeof (char) * buf_size);

    while (read_line (circuit_fp))
    {
      int  wire_num;
      char  gate_name[33];
      char  gate_type[5];
      int  num_fan_out;
      int  num_fan_in;

      /* parse line */
      sscanf (buffer, "%i %s %s %i %i",
          &wire_num,
          gate_name,
          gate_type,
          &num_fan_out,
          &num_fan_in );

      if (!wire_num)
      {
        fprintf (stderr, "Error: wire number 0 is used\n");
        exit (1);
      }

      /* add wire to the wire list */
      {
        if (wire_num+1 >= temp_num_wire)
        {
          temp_num_wire += INI_LIST_SIZE;
          cut->wire_list = (Wire *)
            realloc (cut->wire_list,
                sizeof (Wire) * temp_num_wire);
          if (!cut->wire_list)
          {
            fprintf (stderr, "can't realloc!\n");
            exit(1);
          }
          memset (cut->wire_list + temp_num_wire - INI_LIST_SIZE,
              0, sizeof (Wire) * INI_LIST_SIZE);
        }
        cut->wire_list[wire_num].exist = 1;
        cut->wire_list[wire_num].from_gate = cut->num_gate;
        cut->num_wire = wire_num + 1;
        cut->wire_list[wire_num].node_num = cut->num_gate;
      }

      /* add primary output */
```

```
if (!num_fan_out)
{
  cut->wire_list[wire_num].to_gate = 0;

  if (cut->num_output == temp_num_output)
  {
    temp_num_output += INI_LIST_SIZE;
    cut->output_list = (int  *)
      realloc (cut->output_list,
          sizeof (int ) * temp_num_output);
    if (!cut->output_list)
    {
      fprintf (stderr, "can't realloc!\n");
      exit(1);
    }
  }

  cut->output_list[cut->num_output] = wire_num;
  cut->num_output++;
}

/* add gate */
{
  if (cut->num_gate == temp_num_gate)
  {
    temp_num_gate += INI_LIST_SIZE;
    cut->gate_list = (Gate *)
      realloc (cut->gate_list,
          sizeof (Gate) * temp_num_gate);
    if (!cut->gate_list)
    {
      fprintf (stderr, "can't realloc!\n");
      exit(1);
    }
  }

  cut->gate_list[cut->num_gate].gate_num = wire_num;
  cut->gate_list[cut->num_gate].out_node = cut->num_gate;

  /* input number */
  if (!num_fan_in)
  {
    if (cut->num_input == temp_num_input)
    {
      temp_num_input += INI_LIST_SIZE;
      cut->input_list = (int *)
        realloc (cut->input_list,
            sizeof (int ) * temp_num_input);
      if (!cut->input_list)
      {
        fprintf (stderr, "can't realloc!\n");
        exit(1);
      }
    }
    cut->input_list[cut->num_input] = cut->num_gate;
    cut->gate_list[cut->num_gate].input_num
      = cut->num_input++;
  }

  /* determine gate type */
  if (!strcmp (gate_type, "inpt"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_INPT;
    //cut->gate_list[cut->num_gate].gate    = &g_inpt;
  }

  if (!strcmp (gate_type, "buff"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_BUFF;
    //cut->gate_list[cut->num_gate].gate    = &g_buff;
  }

  if (!strcmp (gate_type, "not"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_NOT;
    //cut->gate_list[cut->num_gate].gate    = &g_not;
  }

  if (!strcmp (gate_type, "and"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_AND;
    //cut->gate_list[cut->num_gate].gate    = &g_and;
  }

  if (!strcmp (gate_type, "nand"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_NAND;
    //cut->gate_list[cut->num_gate].gate    = &g_nand;
  }

  if (!strcmp (gate_type, "or"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_OR;
    //cut->gate_list[cut->num_gate].gate    = &g_or;
  }

  if (!strcmp (gate_type, "nor"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_NOR;
    //cut->gate_list[cut->num_gate].gate    = &g_nor;
  }

  if (!strcmp (gate_type, "xor"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_XOR;
    //cut->gate_list[cut->num_gate].gate    = &g_xor;
  }

  if (!strcmp (gate_type, "xnor"))
  {
    cut->gate_list[cut->num_gate].gate_type = g_XNOR;
    //cut->gate_list[cut->num_gate].gate    = &g_xnor;
  }


  /* deal with fan_in/fan_out */
  cut->gate_list[cut->num_gate].n_fan_in  = num_fan_in;
  cut->gate_list[cut->num_gate].n_fan_out = num_fan_out;

  cut->gate_list[cut->num_gate].in_node =
    (int  *) malloc (sizeof (int ) * num_fan_in);
  cut->gate_list[cut->num_gate].fan_in =
    (int  *) malloc (sizeof (int ) * num_fan_in);
  cut->gate_list[cut->num_gate].fan_out =
    (int  *) malloc (sizeof (int ) * num_fan_out);

  if (!(
    cut->gate_list[cut->num_gate].in_node &&
    cut->gate_list[cut->num_gate].fan_in &&
    cut->gate_list[cut->num_gate].fan_out
    ))
  {
    fprintf (stderr, "can't realloc!\n");
    exit(1);
  }


  if (num_fan_in)
  {
    int  s_wire_num;

    if (!read_line(circuit_fp))
    {
      fprintf (stderr,"Error: cannot read gate input\n");
      exit (1);
    }

    while (num_fan_in)
    {
      if (num_fan_in ==
        cut->gate_list[cut->num_gate].n_fan_in)
        s_wire_num =
          atoi ((char *)strtok(buffer, " "));
      else
        s_wire_num =
          atoi ((char *)strtok(NULL, " "));

      cut->gate_list[cut->num_gate]
        .fan_in[num_fan_in-1] = s_wire_num;

      cut->gate_list[cut->num_gate]
        .in_node[num_fan_in-1]
          = cut->wire_list[s_wire_num].node_num;

      cut->wire_list[s_wire_num].to_gate
        = cut->num_gate;

      num_fan_in--;
    }
  }


  if (num_fan_out == 1)
    cut->gate_list[cut->num_gate].fan_out[0] = wire_num;


  if (num_fan_out > 1)
  {
    cut->wire_list[wire_num].to_gate = 0;
    while (num_fan_out)
    {
      if (!read_line(circuit_fp))
      {
        fprintf (stderr,
            "Error: cannot read gate fan out\n");
        exit (1);
      }

      sscanf (buffer, "%d", &wire_num);


      /* add wire */
      {
        if (wire_num+1 >= temp_num_wire)
        {
          temp_num_wire += INI_LIST_SIZE;
          cut->wire_list = (Wire *)
            realloc (cut->wire_list,
                sizeof (Wire) * temp_num_wire);
          if (!cut->wire_list)
          {
            fprintf (stderr, "can't realloc!\n");
```

```
            exit(1);
          }

          memset (cut->wire_list
            + temp_num_wire - INI_LIST_SIZE,
            0, sizeof (Wire) * INI_LIST_SIZE);
        }
        cut->wire_list[wire_num].exist = 1;
        cut->wire_list[wire_num].from_gate=cut->num_gate;
        cut->num_wire = wire_num + 1;
        cut->wire_list[wire_num].node_num =cut->num_gate;
      }


      cut->gate_list[cut->num_gate].fan_out[
        cut->gate_list[cut->num_gate].n_fan_out
        - num_fan_out]
        * wire_num;

      num_fan_out--;
    }
  }

  cut->num_gate++;
 }
}

free (buffer);
}

fclose (circuit_fp);


/* shrink the lists */
{
  cut->wire_list = (Wire *) realloc (cut->wire_list,
        sizeof (Wire) * cut->num_wire);
  cut->gate_list = (Gate *) realloc (cut->gate_list,
        sizeof (Gate) * cut->num_gate);
  cut->input_list = (int  *) realloc (cut->input_list,
         sizeof (int ) * cut->num_input);
  cut->output_list = (int  *) realloc (cut->output_list,
         sizeof (int ) * cut->num_output);
  if (!(cut->wire_list &&
      cut->gate_list &&
      cut->input_list &&
      cut->output_list))
  {
    fprintf (stderr, "can't realloc!\n");
    exit(1);
  }
}

}



// modifies the global buffer variable

int  read_line (FILE *fp)
{
  int  ch;
  int  count;

  int   buf2_size = BUF_SIZE;
  char *buffer2;

  buffer2 = (char *) malloc (sizeof (char) * buf2_size);

  while (1)
  {
    count = 0;
    while ((ch = fgetc(fp)) != EOF)
    {
      if (ch == '\n') break;
      buffer2[count++] = ch;

      if (count == buf2_size)
      {
        buf2_size += BUF_SIZE;
        buffer2 = (char *) realloc (buffer2,
                    sizeof(char)*buf2_size);
      }
    }

    // skip lines without char
    if (!count && ch != EOF) continue;
    if (buffer2[0] != '*') break;
  }
  buffer2[count] = 0;
/*
  printf ("%s\n", buffer2);
*/

  /* trim line */
  {
    int  i, j;
    char flag = 0;

    for (i=0, j=0; buffer2[i]; i++)
    {
      if (!isspace(buffer2[i]))
      {
        if (j == buf_size)
        {
          buf_size += BUF_SIZE;
          buffer = (char *) realloc (buffer,
                      sizeof(char)*buf_size);
        }
        buffer[j++] = buffer2[i];
        flag = 1;
      }
      else
      {
        if (flag)
        {
          buffer[j++] = ' ';
          flag = 0;
        }
      }
    }
    count = j;
  }
  buffer[count] = 0;
/*
  printf ("%s\n", buffer);
*/

  free (buffer2);
  return count;
}



void Print_Circuit (Circuit cut, int  level)
{
  if (level == 0) return;

  printf ("Maximum number of wire  : %d\n", cut.num_wire);
  printf ("Maximum number of gate  : %d\n", cut.num_gate);
  printf ("Total number of input   : %d\n", cut.num_input);
  printf ("Total number of output  : %d\n", cut.num_output);

  if (level==1) return;

  /* print wire list */
  {
    int  i;
    for (i=0; i<cut.num_wire; i++)
    if (cut.wire_list[i].exist)
    {
      printf ("wire %d - from %d, to %d - node %d\n",
          i,
          cut.wire_list[i].from_gate,
          cut.wire_list[i].to_gate,
          cut.wire_list[i].node_num);
    }
  }

  /* print gate list */
  {
    int  i;
    for (i=0; i<cut.num_gate; i++)
    {
      printf ("gate %d - type %s, output %d(%d)\n",
          i,
          gate_table[cut.gate_list[i].gate_type],
          cut.gate_list[i].gate_num,
          cut.gate_list[i].out_node);
    }
  }


  /* print output list */
  {
    for (int i=0; i<cut.num_output; i++)
    {
      printf ("primary output %d wire %d(%d)\n",
          i,
          cut.output_list[i],
          cut.wire_list[cut.output_list[i]].from_gate);

    }
  }
}



int CacheEval (Circuit *cut)
{
  int maxcache=0;
  int curcache=0;
  char cacheref[cut->num_gate];

  memset (cacheref, 0, cut->num_gate);

  for (int i=0; i<cut->num_gate; i++)
  {
    curcache++;
    cacheref[i] = cut->gate_list[i].n_fan_out;

    if (cut->gate_list[i].gate_type != G_INPT)
```

```
    for (int j=0; j<cut->gate_list[i].n_fan_in; j++)
      if (--cacheref[cut->gate_list[i].in_node[j]] == 0 &&
          cut->gate_list[cut->gate_list[i].in_node[j]]
            .gate_type != g_INPT) curcache--;

    maxcache = (maxcache > curcache)? maxcache : curcache;
  }

  return maxcache;
}

// end of read_iscas.C
```

A.1.5 gen_fault.C

```
/*
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*PAR Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * gen_fault.C
 *
 * Purpose : generate faults from the structure of the circuit under test
 *           perform fault collapsing and fault implication setup as well.
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include "structure.h"
#include "prototype.h"

char *fault_table[8] =
  {"SA0", "SA1", " SR", " SF", " BO", " PO", "iNO", "iPO"};
char *gate_table[10] =
  {"INPT","BUFF","NOT","AND","NAND","OR","NOR","XOR","XNOR","DUM"};

int  num_wire;

void DrawNode (Fault *fault_list, int level);

/*
 * Pre   fault_list is allocated to 6 * cut->num_wire
 *
 * delay faults and stuck-at faults are initially marked as exist
 * whereas stuck-open faults are initially marked as non-exist
 *
 * for intermediate faults for AND/OR gates, the second input pin is used
 * to represent the fault, which would be marked as either iPO or iNO
 *
 * Buffers are not replaced as NOT-NOT structure
 *
 */
void Gen_Fault (Circuit *cut, Fault *fault_list,  char sim_fault)
{
  int i;

  /* remove all the non existing faults */
  num_wire = cut->num_wire;
  for (i=0; i<cut->num_wire; i++)
  {
    if (!cut->wire_list[i].exist)
    {
      a_SA0(i).flag = fd_NONEXIST;
      a_SA1(i).flag = fd_NONEXIST;
      a_SR (i).flag = fd_NONEXIST;
      a_SF (i).flag = fd_NONEXIST;
    }
    else
    {
      a_SA0(i).flag = fd_UNDETECTED;
      a_SA1(i).flag = fd_UNDETECTED;
      a_SR (i).flag = fd_UNDETECTED;
      a_SF (i).flag = fd_UNDETECTED;
    }
    a_PO (i).flag = fd_NONEXIST;
    a_NO (i).flag = fd_NONEXIST;

    a_SA0(i).implied1 = a_SA0(i).implied2 = NULL;
    a_SA1(i).implied1 = a_SA1(i).implied2 = NULL;
              a_SR (i).implied2 = NULL;
              a_SF (i).implied2 = NULL;
    a_NO (i).implied1 = a_NO (i).implied2 = NULL;
    a_PO (i).implied1 = a_PO (i).implied2 = NULL;

    // trival implication
    a_SR (i).implied1 = &(a_SA0(i));
    a_SF (i).implied1 = &(a_SA1(i));

    a_SA0(i).wire_num =
    a_SA1(i).wire_num =
    a_SR (i).wire_num =
    a_SF (i).wire_num =
    a_NO (i).wire_num =
    a_PO (i).wire_num = i;

    a_SA0(i).type = f_SA0;
    a_SA1(i).type = f_SA1;
    a_SR (i).type = f_SR;
    a_SF (i).type = f_SF;
    a_NO (i).type = f_NO;
    a_PO (i).type = f_PO;
  }

  for (i=0 ;i<cut->num_gate; i++)
  {
    int  j;
    Gate fgate;
    Gate tgate = cut->gate_list[i];

    switch (tgate.gate_type)
    {
      case g_INPT :
      break;

      case g_BUFF :
        ////////////////////////////////////////////////////
        //
        //  Buffer (input I, output Z)
        //  fault   first   second  good  faulty
        //     X      I      Z      Z
        //  ---------------------------------------
        //  SA0(I)  x      1      1      0
        //  SA1(I)  x      0      0      1
        //  SA0(Z)  x      1      1      0
        //  SA1(Z)  x      0      0      1
        //
        //  SR(I)   0      1      01     00
        //  SF(I)   1      0      10     11
        //  SR(Z)   0      1      01     00
        //  SF(Z)   1      0      10     11
        //
        //  NO(I)   0      1      01     00
        //  PO(I)   1      0      10     11
        //  ---------------------------------------
        //
        ////////////////////////////////////////////////////

        a_SA0(tgate.fan_in[0]).flag = fd_EQUIVAL;
        a_SA1(tgate.fan_in[0]).flag = fd_EQUIVAL;
        a_SR (tgate.gate_num ).flag = fd_EQUIVAL;
        a_SF (tgate.gate_num ).flag = fd_EQUIVAL;

        a_NO (tgate.fan_in[0]).flag = fd_UNDETECTED;
        a_PO (tgate.fan_in[0]).flag = fd_UNDETECTED;

        a_NO (tgate.fan_in[0]).implied1 = &(a_SA0(tgate.gate_num ));
        a_PO (tgate.fan_in[0]).implied1 = &(a_SA1(tgate.gate_num ));
        a_SR (tgate.fan_in[0]).implied1 = &(a_SA0(tgate.gate_num ));
        a_SF (tgate.fan_in[0]).implied1 = &(a_SA1(tgate.gate_num ));

        // stuck-open eq delay
        a_NO (tgate.fan_in[0]).implied2 = &(a_SR (tgate.fan_in[0]));
        a_PO (tgate.fan_in[0]).implied2 = &(a_SF (tgate.fan_in[0]));
        a_SR (tgate.fan_in[0]).implied2 = &(a_NO (tgate.fan_in[0]));
        a_SF (tgate.fan_in[0]).implied2 = &(a_PO (tgate.fan_in[0]));

      break;

      case g_NOT :
        ////////////////////////////////////////////////////
        //
        //  Not Gate  (input I, output Z)
        //  fault   first   second  good  faulty
        //     X      I      Z      Z
        //  ---------------------------------------
        //  SA0(I)  x      1      0      1
        //  SA1(I)  x      0      1      0
        //  SA0(Z)  x      0      1      0
        //  SA1(Z)  x      1      0      1
        //
        //  SR(I)   0      1      10     11
        //  SF(I)   1      0      01     00
        //  SR(Z)   1      0      01     00
        //  SF(Z)   0      1      10     11
        //
        //  NO(I)   0      1      10     11
        //  PO(I)   1      0      01     00
        //  ---------------------------------------
        //
        ////////////////////////////////////////////////////

        a_SA0(tgate.fan_in[0]).flag = fd_EQUIVAL;
        a_SA1(tgate.fan_in[0]).flag = fd_EQUIVAL;
```

```
  a_SR (tgate gate_num ) flag = fd_EQUIVAL;
  a_SF (tgate gate_num ) flag = fd_EQUIVAL;

  a_NO (tgate fan_in[0]) flag = fd_UNDETECTED;
  a_PO (tgate fan_in[0]) flag = fd_UNDETECTED;

  a_NO (tgate fan_in[0]) implied1 = #(a_SA1(tgate gate_num ));
  a_PO (tgate fan_in[0]) implied1 = #(a_SA0(tgate gate_num ));
  a_SR (tgate fan_in[0]) implied1 = #(a_SA1(tgate gate_num ));
  a_SF (tgate fan_in[0]) implied1 = #(a_SA0(tgate gate_num ));

  // stuck-open eq delay
  a_NO (tgate fan_in[0]) implied2 = #(a_SR (tgate fan_in[0]));
  a_PO (tgate fan_in[0]) implied2 = #(a_SF (tgate fan_in[0]));
  a_SR (tgate fan_in[0]) implied2 = #(a_NO (tgate fan_in[0]));
  a_SF (tgate fan_in[0]) implied2 = #(a_PO (tgate fan_in[0]));

  // special collapsing for stuck-open
  fgate=cut->gate_list[
    cut->wire_list[tgate fan_in[0]].from_gate ];
  if (fgate.n_fan_out == 1)
  {
    switch (fgate.gate_type)
    {
      case g_NAND :
        a_PO(tgate fan_in[0]).flag = fd_EQUIVAL;
      break;

      case g_NOR :
        a_NO(tgate fan_in[0]).flag = fd_EQUIVAL;
      break;

      case g_NOT :
      case g_AND :
      case g_OR :
        a_NO(tgate fan_in[0]).flag = fd_EQUIVAL;
        a_PO(tgate fan_in[0]).flag = fd_EQUIVAL;
      break;

      default : break;
    }
  }

break;


case g_AND  :
  ////////////////////////////////////////////////////
  //
  //  And Gate (input I1 I2, output Z)
  //  fault   first   second  good  faulty
  //      I1I2  I1I2  Z     Z
  //  ---------------------------------------
  //  SA0(I1) xx    11    1     0
  //  SA1(I1) xx    01    0     1
  //  SA0(I2) xx    11    1     0
  //  SA1(I2) xx    10    0     1
  //  SA0(Z)  xx    11    1     0
  //  SA1(Z)  xx    (*1)  0     1
  //
  //  SR(I1)  0x    11    01    00
  //  SF(I1)  1x    01    x0    x1
  //  SR(I2)  x0    11    01    00
  //  SF(I2)  1x    10    x0    x1
  //  SR(Z)   (*1)  11    01    00
  //  SF(Z)   11    (*1)  10    11
  //
  //  NO(I1)  (*1)  11    01    00
  //  PO(I1)  11    01    10    11
  //  NO(I2)  (*1)  11    01    00
  //  PO(I2)  11    10    10    11
  //  NO(Y)   11    (*1)  10    11
  //  PO(Y)   (*1)  11    01    00
  //  ---------------------------------------
  //
  ////////////////////////////////////////////////////

  a_SR (tgate gate_num ) flag = fd_EQUIVAL;

  a_NO (tgate fan_in[0]) flag = fd_UNDETECTED;
  a_NO (tgate fan_in[0]) implied1 = #(a_SA0(tgate gate_num ));

  for (j=0; j<tgate.n_fan_in; j++)
  {
  a_SA0(tgate fan_in[j]) flag = fd_EQUIVAL;
  a_PO (tgate fan_in[j]) flag = fd_UNDETECTED;
  a_PO (tgate fan_in[j]) implied2 = #(a_NO (tgate fan_in[1]));

  a_PO (tgate fan_in[j]) implied1 = #(a_SF (tgate fan_in[j]));
  a_SF (tgate fan_in[j]) implied1 = #(a_SA1(tgate fan_in[j]));
  a_SA1(tgate fan_in[j]) implied1 = #(a_SA1(tgate gate_num ));

  a_SR (tgate fan_in[j]) implied1 = #(a_NO (tgate fan_in[0]));
  }

  // intermediate fault
  a_NO (tgate fan_in[1]) flag = fd_UNDETECTED;
  a_NO (tgate fan_in[1]) type = f_NO;
  a_NO (tgate fan_in[1]) wire_num = tgate gate_num ;

  a_NO (tgate fan_in[1]) implied1 = #(a_SA1(tgate gate_num ));
  a_NO (tgate fan_in[1]) implied2 = #(a_SF (tgate gate_num ));
  a_SF (tgate gate_num ) implied2  = #(a_NO (tgate fan_in[1]));

break;


case g_NAND :
  ////////////////////////////////////////////////////
  //
  //  Nand Gate (input I1 I2, output Z)
  //  fault   first   second  good  faulty
  //      I1I2  I1I2  Z     Z
  //  ---------------------------------------
  //  SA0(I1) xx    11    0     1
  //  SA1(I1) xx    01    1     0
  //  SA0(I2) xx    11    0     1
  //  SA1(I2) xx    10    1     0
  //  SA0(Z)  xx    (*1)  1     0
  //  SA1(Z)  xx    11    0     1
  //
  //  SR(I1)  0x    11    10    11
  //  SF(I1)  1x    01    x1    x0
  //  SR(I2)  x0    11    10    11
  //  SF(I2)  x1    10    x1    x0
  //  SR(Z)   11    (*1)  01    00
  //  SF(Z)   (*1)  11    10    11
  //
  //  NO(I1)  (*1)  11    10    11
  //  PO(I1)  11    01    01    00
  //  NO(I2)  (*1)  11    10    11
  //  PO(I2)  11    10    01    00
  //  ---------------------------------------
  //
  ////////////////////////////////////////////////////

  a_SF (tgate gate_num ) flag = fd_EQUIVAL;

  a_NO (tgate fan_in[0]) flag = fd_UNDETECTED;
  a_NO (tgate fan_in[0]) implied1 = #(a_SA1(tgate gate_num ));

  for (j=0; j<tgate.n_fan_in; j++)
  {
  a_SA0(tgate fan_in[j]) flag = fd_EQUIVAL;
  a_PO (tgate fan_in[j]) flag = fd_UNDETECTED;
  a_PO (tgate fan_in[j]) implied2 = #(a_SR (tgate gate_num ));

  a_PO (tgate fan_in[j]) implied1 = #(a_SF (tgate fan_in[j]));
  a_SF (tgate fan_in[j]) implied1 = #(a_SA1(tgate fan_in[j]));
  a_SA1(tgate fan_in[j]) implied1 = #(a_SA0(tgate gate_num ));

  a_SR (tgate fan_in[j]) implied1 = #(a_NO (tgate fan_in[0]));
  }
break;


case g_OR
  ////////////////////////////////////////////////////
  //
  //  Or Gate (input I1 I2, output Z)
  //  fault   first   second  good  faulty
  //      I1I2  I1I2  Z     Z
  //  ---------------------------------------
  //  SA0(I1) xx    10    1     0
  //  SA1(I1) xx    00    0     1
  //  SA0(I2) xx    01    1     0
  //  SA1(I2) xx    00    0     1
  //  SA0(Z)  xx    (*0)  1     0
  //  SA1(Z)  xx    00    0     1
  //
  //  SR(I1)  0x    10    x1    x0
  //  SF(I1)  1x    00    10    11
  //  SR(I2)  x0    01    x1    x0
  //  SF(I2)  x1    00    10    11
  //  SR(Z)   00    (*0)  01    00
  //  SF(Z)   (*0)  00    10    11
  //
  //  PO(I1)  (*0)  00    10    11
  //  NO(I1)  00    10    01    00
  //  PO(I2)  (*0)  00    10    11
  //  NO(I2)  00    01    01    00
  //  PO(Y)   00    (*0)  01    00
  //  NO(Y)   (*0)  00    10    11
  //  ---------------------------------------
  //
  ////////////////////////////////////////////////////

  a_SF (tgate gate_num ) flag = fd_EQUIVAL;

  a_PO (tgate fan_in[0]) flag = fd_UNDETECTED;
  a_PO (tgate fan_in[0]) implied1 = #(a_SA1(tgate gate_num ));

  for (j=0; j<tgate.n_fan_in; j++)
  {
  a_SA1(tgate fan_in[j]) flag = fd_EQUIVAL;
  a_NO (tgate fan_in[j]) flag = fd_UNDETECTED;
  a_NO (tgate fan_in[j]) implied2 = #(a_PO (tgate fan_in[1]));

  a_NO (tgate fan_in[j]) implied1 = #(a_SR (tgate fan_in[j]));
  a_SR (tgate fan_in[j]) implied1 = #(a_SA0(tgate fan_in[j]));
  a_SA0(tgate fan_in[j]) implied1 = #(a_SA0(tgate gate_num ));

  a_SF (tgate fan_in[j]) implied1 = #(a_PO (tgate fan_in[0]));
  }
```

```
      // intermediate fault
      a_PO (tgate.fan_in[i]).flag = fd_UNDETECTED;
      a_PO (tgate.fan_in[i]).type = f_iPO;
      a_PO (tgate.fan_in[i]).wire_num = tgate.gate_num ;

      a_PO (tgate.fan_in[i]).implied1 = &(a_SA0(tgate.gate_num ));
      a_PO (tgate.fan_in[i]).implied2 = &(a_SR (tgate.gate_num ));
      a_SR (tgate.gate_num ).implied2  = &(a_PO (tgate.fan_in[i]));

    break;


    case g_XOR  :
      ////////////////////////////////////////////////
      //
      //  Xor Gate (input X1 X2, output Z)
      //  fault    first    second  good  faulty
      //     X1X2  X1X2  Z     Z
      //  ------------------------------------
      //  SA0(X1) xx    10    0     1
      //  SA1(X1) xx    00    1     0
      //  SA0(X2) xx    01    0     1
      //  SA1(X2) xx    00    1     0
      //  SA0(Z)  xx    00    1     0
      //  SA1(Z)  xx    (!0)  0     1
      //
      //  SR(X1)  0x    10    x0    x1
      //  SF(X1)  1x    00    01    00
      //  SR(X2)  x0    01    x0    x1
      //  SF(X2)  x1    00    01    00
      //  SR(Z)   (!0)  00    01    00
      //  SF(Z)   00    (!0)  10    11
      //
      //  PO(X1)  (!0)  00    01    00
      //  NO(X1)  00    10    10    11
      //  PO(X2)  (!0)  00    01    00
      //  NO(X2)  00    01    10    11
      //  ------------------------------------
      //
      ////////////////////////////////////////////////

      a_SR (tgate.gate_num ).flag = fd_EQUIVAL;

      a_PO (tgate.fan_in[0]).flag = fd_UNDETECTED;
      a_PO (tgate.fan_in[0]).implied1 = &(a_SA0(tgate.gate_num ));

      for (j=0; j<tgate.n_fan_in; j++)
      {
      a_SA1(tgate.fan_in[j]).flag = fd_EQUIVAL;
      a_NO (tgate.fan_in[j]).flag = fd_UNDETECTED;
      a_NO (tgate.fan_in[j]).implied2 = &(a_SF (tgate.gate_num ));

      a_NO (tgate.fan_in[j]).implied1 = &(a_SR (tgate.fan_in[j]));
      a_SR (tgate.fan_in[j]).implied1 = &(a_SA0(tgate.fan_in[j]));
      a_SA0(tgate.fan_in[j]).implied1 = &(a_SA1(tgate.gate_num ));

      a_SF (tgate.fan_in[j]).implied1 = &(a_PO (tgate.fan_in[0]));
      }
    break;


    case g_XOR  :
    break;


    case g_XNOR :
    break;

    default  : break;
  }
 }

}


int  count_array[8];

// restructure the linear fault list headed by fault_head
int  Restruct_Fault_List (Fault **fault_head, Fault *fault_list)
{
  Fault *car, *prev;
  Fault temp_head;
  int  count=0;

  memset (count_array, 0, sizeof (int ) * 8);

  if (*fault_head)
  {
    prev = &temp_head;
    for (car = *fault_head; car != NULL; car = car->next)
    {
      if (!car->flag)
      {
        prev->next = car;
        prev = car;
        count++;
        count_array[car->type]++;
      }
    }
  }
  else
  {
    car = &temp_head;
    for (int i=0; i<6*num_wire; i++)
    {
      if (!fault_list[i].flag)
      {
        car->next = fault_list + i;
        car = car->next;
        count++;
        count_array[car->type]++;
      }
    }
    car->next = NULL;
  }
  *fault_head = temp_head.next;

  return count;
}


FaultFamily *Gen_Fault_Family (Circuit *cut, Fault *fault_list)
{
  InitFamily (cut);

  // doesn't work for xor/xnor
  int ctable[9] = {0,1,0,1,0,1,0,1,0};

  for (int i=0; i<cut->num_gate; i++)
  {
    /*
    printf ("gate number %d(%s) with %d fan-ins\n",
      i, gate_table[cut->gate_list[i].gate_type],
      cut->gate_list[i].n_fan_in);
    */
    Gate tgate = cut->gate_list[i];

    // outputs
    for (int j=5; j>=0 ; j--)
    {
      Fault *fault = fault_list + 6*tgate.gate_num + j;
      if (!fault->flag)
      {
        switch (fault->type)
        {
          case f_iPO:
          case f_iNO:
          case f_NO:
          case f_PO:
            // does not belong to this gate
          break;

          case f_SA0:
          case f_SR:
          {
            //printf ("%d\n", fault - fault_list);
            AddToFamily (fault, i, 0);
          }
          break;

          case f_SA1:
          case f_SF:
          {
            //printf ("%d\n", fault - fault_list);
            AddToFamily (fault, i, 1);
          }
          break;
        }
      }
    }


    // take care of iNO and iPO
    if (tgate.n_fan_in > 1)
    {
      if (a_NO(tgate.fan_in[1]).type == f_iNO)
      {
        Fault *fault = fault_list + 6*tgate.fan_in[1] + 0;
        if (fault->flag) break;
        AddToFamily (fault, i, 1);
      }

      if (a_PO(tgate.fan_in[1]).type == f_iPO)
      {
        Fault *fault = fault_list + 6*tgate.fan_in[1] + 1;
        if (fault->flag) break;
        AddToFamily (fault, i, 0);
      }
    }

    // inputs
    for (int j=0; j<tgate.n_fan_in; j++)
    {
      //printf("next input : %d (%d)\n", tgate.in_node[j]);

      if (cut->gate_list[tgate.in_node[j]].n_fan_out == 1)
      {
        // add the NO and PO from this wire
        for (int k=1; k>=0; k--)
        {
          Fault *fault = fault_list + 6*tgate.fan_in[j] + k;
          if (!fault->flag)
          {
```

```
            if (fault->type == f_NO)
            {
              //printf ("%d\n", fault - fault_list);
              AddToFamily (fault, i,
                ctable[tgate.gate_type]);
            }
            else if (fault->type == f_PO)
            {
              //printf ("%d\n", fault - fault_list);
              AddToFamily (fault, i,
                ctable[tgate.gate_type]);
            }
          }
        }
      }
      else
      {
        for (int k=5; k>=0; k--)
        {
          Fault *fault = fault_list + 6*tgate.fan_in[j] + k;
          if (!fault->flag)
          {
            switch (fault->type)
            {
              case f_iPO:
              case f_iNO:
                // already taken care of
              break;

              case f_SA0:
              case f_SR:
              case f_NO:
              {
                //printf ("%d\n", fault - fault_list);
                AddToFamily (fault, i,
                  ctable[tgate.gate_type]);
              }
              break;

              case f_SA1:
              case f_SF:
              case f_PO:
              {
                //printf ("%d\n", fault - fault_list);
                AddToFamily (fault, i,
                  ctable[tgate.gate_type]);
              }
              break;
            }
          }
        }
      }
    }
  }

  FaultFamily *ff = GetFaultFamily();

  for (int i=0; i<cut->num_gate; i++)
  {
    for (int j=0; j<2; j++)
    {
      Fault *car;
      for (car = ff[(i<<1)+j].Member; car != NULL; car = car->next)
      {
        car->gate_num = i;
        car->family   = j;
      }
    }
  }

  return ff;
}


void Print_Fault_List (Fault *fault_list, Fault *fault_head)
{
  int  i;
  int  n=0, p=0;
  Fault *car;

  for (car = fault_head; car != NULL; car = car->next)
  {
    printf ("%d(%d) - ", car->type, car->wire_num);
    printf ("flag = %d, ", car->flag);
    if (car->implied1) printf ("i1 = %d(%d) ", car->implied1->type,
                               car->implied1->wire_num);
    if (car->implied2) printf ("i2 = %d(%d) ", car->implied2->type,
                               car->implied2->wire_num);
    printf ("\n");
  }
}



void Get_Count (int  *n_sa, int  *n_d, int  *n_so)
{
  *n_sa = count_array[f_SA1] + count_array[f_SA0];
  *n_d  = count_array[f_SR ] + count_array[f_SF ];
  *n_so = count_array[f_NO ] + count_array[f_iNO] +
      count_array[f_PO ] + count_array[f_iPO];
}


// some how, this routine doesn't count correctly ...
void Count_Fault_Head
  (Fault *fault_head, int  *n_sa, int  *n_d, int  *n_so)
{
  Fault *car;

  memset (count_array, 0, sizeof (int ) * 8);

  for (car=fault_head; car!=NULL; car=car->next) count_array[car->type]++;

  Get_Count (n_sa, n_d, n_so);
}



void Count_Fault_List(Fault *fault_list, int  *n_sa, int  *n_d, int  *n_so)
{
  int  i;


  memset (count_array, 0, sizeof (int ) * 8);

  for (i=0; i<6*num_wire; i++)
    if (!fault_list[i].flag) count_array[fault_list[i].type]++;

  Get_Count (n_sa, n_d, n_so);
}



void Print_Fault(Fault *fault_list, int  level)
{
  int  i,j;
  Fault car;

  switch (level)
  {
    case 1:
      for (j=0; j<6; j++) for (i=0; i<num_wire; i++)
      {
        car = fault_list[i*6+j];
        if (!car.flag)
          printf ("%s(%d)\n",
               fault_table[car.type],
               car.wire_num);
      }
    break;

    case 2:
      for (j=0; j<6; j++) for (i=0; i<num_wire; i++)
      {
        car = fault_list[i*6+j];
        if (car.flag != fd_NONEXIST)
        {
          printf ("%s(%d) - ",
            fault_table[car.type], car.wire_num);
          if (car.flag == fd_UNDETECTED)
            printf ("Undetected ");
          if (car.flag == fd_EQUIVAL)
            printf ("Collapsed ");
          if (car.flag == fd_DETECTED)
            printf ("Detected ");
          if (car.flag == fd_IMPLIED)
            printf ("Detected by Implication ");
          printf ("(%x)",car.flag);
          printf ("\n");
        }
      }
    break;

    case 3: // drawtree
    {
      int nRoot = 0;
      Fault *fault_head = NULL;
      Fault *car;
      Restruct_Fault_List (&fault_head, fault_list);
      for (car = fault_head; car != NULL; car = car->next)
      {
        if (!car->flag)
        {
          printf ("Node #%d\n",++nRoot);
          DrawNode (car, 0);
        }
      }
      for (car = fault_head; car != NULL; car = car->next)
      {
        car->flag &= !fd_DEBUG;  // clear the debugging flag
      }
      printf ("Total number of Roots = %d\n", nRoot);
      int nsa, nso, nd;
      Get_Count (&nsa, &nd, &nso);
      printf ("Total number of fault = %d\n", nsa+nd+nso);
      printf ("Maximum Efficiency = %3.2f%%\n",
        nRoot*100.0/(nsa+nd+nso));
    }
    break;

    case 4:
      PrintFaultFamily();
    break;
```

```
  }

}


void DrawNode (Fault *fault, int level)
{
  fault->flag |= fd_DEBUG;
  for (int i=0; i<level; i++) printf ("|");
  printf ("~ %s(%d) [%d %d] (flag=%d)\n",
    fault_table[fault->type],
    fault->wire_num,
    fault->gate_num,
    fault->family,
    fault->flag);

  if (fault->implied1)
    if (fault->implied1->flag == fd_UNDETECTED)
      DrawNode (fault->implied1, level+1);

  if (fault->implied2)
    if (fault->implied2->flag == fd_UNDETECTED)
      DrawNode (fault->implied2, level+1);

  return;
}

// end of gen_fault.C
```

A.1.6 fault_family.C

```
/*
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the CoRAM Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * fault_family.C
 *
 * Purpose : a set of routines for dealing with fault families.
 *
 * Compile Options :
 *   FPSDFS - enable depth first search fault grouping
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>

#include "structure.h"
#include "prototype.h"

FaultFamily *ff;
int num_gate;
int n_fault;

char *checklist;
int  *dfs;

/////////////
// Recursive Depth First Search
/////////////
int R_DFS(Circuit *cut, int gate_num, int index)
{
  if (checklist[gate_num]) return index;
  if (cut->gate_list[gate_num].n_fan_in)
  {
    for (int i=0; i<cut->gate_list[gate_num].n_fan_in; i++)
    {
      int from_gate = cut->wire_list[
                cut->gate_list[gate_num].fan_in[i]
              ].from_gate;

      index = R_DFS(cut, from_gate, index);
    }
  }
  checklist[gate_num] = 1;
  dfs[index] = gate_num;
  //printf ("%d added %d\n", index, gate_num);
  return index+1;
}

void DepthFirstSearch(Circuit *cut)
{
  checklist[0] = 1;
  dfs[0] = 0;
  int index = 1;

  for (int i=0; i<cut->num_output; i++)
  {
    int from_gate = cut->wire_list[cut->output_list[i]].from_gate;

    //printf ("output %d %d\n",i,from_gate);
    index = R_DFS(cut, from_gate, index);
  }
}


void InitFamily  (Circuit *cut)
{
  ff = (FaultFamily *) malloc (sizeof (FaultFamily) * 2 * cut->num_gate);
  num_gate = cut->num_gate;

  for (int i=0; i<cut->num_gate; i++)
  {
    int index0 = (i<<1);
    int index1 = (i<<1)+1;

    ff[index0].gate_num = i;
    ff[index0].type     = f_SA0;
    ff[index0].Member   = NULL;

    ff[index1].gate_num = i;
    ff[index1].type     = f_SA1;
    ff[index1].Member   = NULL;
  }

  n_fault = 0;

#ifdef FPSDFS
  dfs = (int *) malloc (sizeof(int) * cut->num_gate);
  checklist = (char *) malloc (sizeof(char) * cut->num_gate);
  memset (dfs, 0, sizeof(int) * cut->num_gate);
  memset (checklist, 0, sizeof(char) * cut->num_gate);

  DepthFirstSearch(cut);
#endif

}

void AddToFamily (Fault *fault, int gate_num, int type)
{
  int index = (gate_num << 1) + type;

  fault->next = ff[index].Member;
  ff[index].Member = fault;

  /*
  printf ("link : ");
  for (Fault *car = ff[index].Member; car != NULL; car = car->next)
  {
    printf ("%s[%d] ", fault_table[car->type], car->wire_num);
  }
  printf ("\n\n");
  */
}


void TransferFamily (int from_gate_num, int from_type,
          int to_gate_num, int to_type)
{
  //printf ("Transfering from %d to %d\n", from_gate_num, to_gate_num);

  int to_index   = (to_gate_num << 1) + to_type;
  int from_index = (from_gate_num << 1) + from_type;

  Fault *car;

  car = ff[from_index].Member;
  if (car)
  {
    while (car->next != NULL) car = car->next;

    car->next = ff[to_index].Member;
    ff[to_index].Member = ff[from_index].Member;
    ff[from_index].Member = NULL;
  }
}


FaultFamily *GetFaultFamily()
{
  FaultFamily head;
  FaultFamily *car = &head;
  int count = 0;

  for (int i=0; i<num_gate; i++)
  {
    for (int j=0; j<2; j++)
    {
#ifdef FPSDFS
      int index = (dfs[i]<<1) + j;
#else
      int index = (i<<1) + j;
#endif

      if (ff[index].Member != NULL)
```

```
      {
        car->next = ff + index;
        car->next->prev = car;
        car = car->next;
        count++;
      }
    }
  }
  car->next = NULL;
  head.next->prev = NULL;

  //printf ("Number of Valid Families = %d\n", count);
  return head.next;
}


void PrintFaultFamily()
{
  int count = 0;
  FaultFamily *ff;

  for (int i=0; i<num_gate; i++)
  {
    for (int j=0; j<2; j++)
    {
      int index = (i<<1) + j;
      int localcount = 0;

      printf ("Fault Family : %d(%d)\n", i, j);
      printf ("  ");
      for (Fault *carcar = ff[index].Member;
         carcar != NULL;
         carcar = carcar->next)
      {
        printf ("%s(%d) ",fault_table[carcar->type],
            carcar->wire_num);
      }
      printf ("\n");

      for (Fault *carcar = ff[index].Member;
         carcar != NULL;
         carcar = carcar->next)
      {
        if (!carcar->flag) DrawNode (carcar,0);
        localcount++;
      }

      printf ("Faults in this family = %d\n\n",localcount);
      count += localcount;
    }
  }

  for (int i=0; i<num_gate; i++)
  {
    for (int j=0; j<2; j++)
    {
      int index = (i<<1) + j;

      for (Fault *carcar = ff[index].Member;
         carcar != NULL;
         carcar = carcar->next)
      {
        carcar->flag &= !fd_DEBUG;
      }
    }
  }

  printf ("total number of faults in Family = %d\n", count);
}



////////////////////////////////
//
// FaultGroup
//
// when fault group is used, the link list feature in the
// fault family implementation can be ignored, because the
// link-list feature of FaultGroup will take over.
//
// Each FaultGroup contains two fault families, each fault
// family contains a list of Faults as members.  When all
// the member of a fault family are detected, the pointer
// in the FaultGroup to this family will point to NULL.
// When both of these pointers are NULL in a FaultGroup,
// the FaultGroup should be dropped from the DFS sorted list.
//
////////////////////////////////

FaultGroup *FG;       // array of fault groups
FaultGroup DFS_head;    // dummy head pointer to DFS sorted FG
int    g_n_fg;

// return the address of DFS_head
FaultGroup *InitFaultGroup(Circuit *cut, Fault *fault_list)
{
  //InitFamily(cut);
  Gen_Fault_Family(cut, fault_list);

  FG = (FaultGroup *) malloc (sizeof(FaultGroup) * cut->num_gate);

  for (int i=0; i<cut->num_gate; i++)
  {
    FG[i].gate_num = i;
    for (int j=0; j<2; j++)
    {
      if (ff[(i<<1)+j].Member != NULL)
        FG[i].FF[j] = &ff[(i<<1) + j];
      else
        FG[i].FF[j] = NULL;
      FG[i].TrigFault[j] = NULL;
      FG[i].TrigFlag[j] = NULL;
    }
  }

#ifdef FPSDFS
  // construct the DFS sorted FaultGroup List
  FaultGroup *car = &DFS_head;
  car->dfs_prev = NULL;
  for (int i=0; i<cut->num_gate; i++)
  {
    car->dfs_next = FG + dfs[i];
    car->dfs_next->dfs_prev = car;
    car = car->dfs_next;
    g_n_fg++;
  }
  car->dfs_next = NULL;
#else
  // construct a simple FaultGroup List
  FaultGroup *car = &DFS_head;
  car->dfs_prev = NULL;
  for (int i=0; i<cut->num_gate; i++)
  {
    car->dfs_next = FG + i;
    car->dfs_next->dfs_prev = car;
    car = car->dfs_next;
    g_n_fg++;
  }
  car->dfs_next = NULL;
#endif

  return &DFS_head;
}

bool CleanFaultGroup(FaultGroup *fg)
{
  bool empty = true;

  for (int i=0; i<2; i++)
  {
    if (fg->FF[i])
    {
      // first move the member list somewhere else
      Fault *car = fg->FF[i]->Member;
      fg->FF[i]->Member = NULL;
      Fault *mcar = NULL;

      // push back the fault if it's not detected
      for (; car != NULL; car = car->next)
      {
        if (car->flag == fd_UNDETECTED)
        {
          if (!fg->FF[i]->Member) fg->FF[i]->Member = car;
          else          mcar->next = car;
          mcar = car;
        }
      }

      // terminate Member list if exist
      // otherwise set fault family pointer to NULL
      if (mcar) {mcar->next = NULL; empty = false;}
      else      fg->FF[i] = NULL;
    }
  }

  if (empty)
  {
    // remove fault group from the DFS_head list
    fg->dfs_prev->dfs_next = fg->dfs_next;
    if (fg->dfs_next) fg->dfs_next->dfs_prev = fg->dfs_prev;
    g_n_fg--;
    return true;
  }
  return false;
}

int GetNumFaultGroup ()
{
  return g_n_fg;
}

// end of fault_family.C
```

A.1.7 heap.h

```
/*
 *
 * ..................................
 *
 * Parallel Fault Simulators on the C*RAN Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L -C  Kwong, Edmonton, Canada
```

```
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * heap.h
 * interface to the heap.C module.
 */

extern void new_heap();
extern void add_heap(int  ele);
extern int  top_heap();
extern int  pop_heap();
extern void ini_heap();
extern void print_heap();
extern char check_heap();
extern int  read_heap();

// end of heap.h
```

A.1.8 heap.C

```
/*
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*RAY Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * heap.C
 *
 * Purpose : implements an int heap for sorting gate evaluation events.
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include "heap.h"

int  *heap;
int  heap_size;
int  heap_count;

#define inc_size 1024

void new_heap()
{
  heap_count = 0;
}


int  top_heap()
{
  return heap_count?heap[0]:-1;
}


#define left   ((car)*2+1)
#define right  ((car)*2+2)
#define parent ((car-1)>>1)

void add_heap(int  ele)
{
  int  car;
  int  temp;

  car = heap_count++;

  if (heap_count > heap_size)
  {
    heap_size += inc_size;
    heap = (int  *) realloc (heap, sizeof(int )*heap_size);
  }

  heap[car] = ele;
  while (parent>=0 && heap[car] < heap[parent])
  {
    temp = heap[parent];
    heap[parent] = heap[car];
    heap[car] = temp;
    car = parent;
  }
}


int  pop_heap()
{
  if (heap_count == 0) return -1;

  int  head = heap[0];
  int  car  = 0;
  int  temp = heap[--heap_count];

  for (;;)
  {
    if (left >= heap_count)
    {
      heap[car] = temp;
      return head;
    }

    if (right >= heap_count)
    {
      if (heap[left] > temp)
      {
         heap[car] = temp;
         return head;
      }
      else
      {
          heap[car] = heap[left];
          heap[left] = temp;
          return head;
      }
    }


    if (temp < heap[left] && temp < heap[right])
    {
      heap[car] = temp;
      return head;
    }
    else
    {
      if (heap[left] < heap[right])
      {
        heap[car] = heap[left];
        car = left;
        continue;
      }
      else
      {
        heap[car] = heap[right];
        car = right;
        continue;
      }
    }
  }
}


void ini_heap()
{
  if (heap) free (heap);
  heap_size = inc_size;
  heap = (int  *) malloc (sizeof (int ) * heap_size);
  heap_count = 0;
}

void print_heap()
{
  int  car;
  for (car = 0; car < heap_count; car++)
    printf ("%d ", heap[car]);
  printf ("\n");
}

char check_heap()
{
  char fault = 0;
  int  car;
  for (car=0; car<(heap_count-1)>>1; car++)
  {
    if (heap[left] < heap[car])
    {
      printf ("check left  %d-%d\n",heap[car],heap[left]);
      fault = 1;
    }
    if (heap[right] < heap[car])
    {
      printf ("check right %d-%d\n",heap[car],heap[right]);
      fault = 1;
    }
  }
  return fault;
}


// simply shorten the heap by returning the last value
```

```
int  read_heap()
{
  return heap_count?heap[--heap_count]:0;
}

// end of heap.C
```

A.1.9 cram_mapper.h

```
/*

 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*RAN Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * cram_mapper.h
 *   using cram_mapper functions may destroy C*RAN register values.
 */

/* cram_mapper.C */
extern void Init_CRAN_Mapper (Circuit *cut, int *node_value);
extern void Init_CRAN_Mapper (Circuit *cut, cint *node_value);
extern void Free_All_Node ();
extern cboolean *Obtain_Node (int node);
extern cboolean *Obtain_Node_w_Loading (int node);
extern cboolean *Obtain_Node_f_mapper (int node);
extern cboolean *Use_Node (int node);
extern void Gate_Eval(Circuit *cut, int node);

// end of cram_mapper.h
```

A.1.10 cram_mapper.C

```
/*

 * cram_mapper.C
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*RAN Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *
 * Purpose : to save cram memory, results of gate evaulations are kept
 *           for as long as it is needed for further gate evaulation.
 *           This module keeps track of the mapings from circuit nodes
 *           to cram cbooleans and returns pointers to cbooleans for
 *           operation.
 *
 * Compile Options :
 *   CM_PURE - load fault-free values from C*RAN instead of int array
 */

#include <stdio.h>
#include <stdlib.h>
#include <cram.h>

#include "structure.h"
#include "prototype.h"
#include "cram_mapper.h"

typedef struct CRANNode_struct CRANNode;
struct CRANNode_struct
{
  cboolean  data;

  CRANNode *next;
  CRANNode *prev;
};

CRANNode *avail;
CRANNode *temp_node;

CRANNode **node_mapper;
int *max_ref;       // array of max gate_num that refers to a gate
int current_gate_num;   // current gate number being evaluated

#ifdef CM_PURE
cint *g_node_value;   // fault-free simulation's node_value
#else
int *g_node_value;    // fault-free simulation's node_value
#endif


/////
// g_n_pat is the number of patterns in simulation, which is a power of 2
/////

int g_n_pat;

static Circuit *g_cut;

#ifdef CM_PURE
void Init_CRAN_Mapper (Circuit *cut, cint *node_value)
#else
void Init_CRAN_Mapper (Circuit *cut, int *node_value)
#endif
{
  avail = NULL;
  temp_node = NULL;

  node_mapper =
    (CRANNode **) malloc (sizeof(CRANNode *) * cut->num_gate);
  memset (node_mapper, 0, sizeof (CRANNode *) * cut->num_gate);

  // this part could be moved to read_iscas.C for module integraty
  max_ref = (int *) malloc (sizeof (int) * cut->num_gate);
  memset (max_ref, 0, sizeof (int) * cut->num_gate);
  for (int i=0; i<cut->num_gate; i++)
  {
    if (cut->gate_list[i].n_fan_out == 0)
    {
      // primary output
      max_ref[i] = cut->num_gate;
    }
    else
    {
      for (int j=0; j<cut->gate_list[i].n_fan_out; j++)
      {
        int out_gate = cut->wire_list[
                  cut->gate_list[i].fan_out[j]
                ].to_gate;
        max_ref[i] = max_ref[i] > out_gate?
              max_ref[i] : out_gate;
      }
    }
  }

  g_cut = cut;
  g_node_value = node_value;
}

void Free_All_Node()
{
  // simply make all the nodes become available

  if ( CRANNode *car = temp_node )
  {
    while (car->next)
    {
      //printf ("yiks (%d)!\n",car);
      car = car->next;
    }
    car->next = avail;
    avail = temp_node;
    temp_node = NULL;

    memset (node_mapper, 0, sizeof(CRANNode *) * g_cut->num_gate);
  }
}


inline Load_Node (int node, cboolean &data)
{
  #ifdef CM_PURE
  g_node_value->operate (node, op_m, writey);
  data.operate (0, op_y, groupwrite);
  #else
  int ffree = g_node_value[node];

  if (ffree & 1) data.operate (0, op_one, groupwrite);
  else       data.operate (0, op_zero, groupwrite);
  #endif
}


inline CRANNode *alloc_Node (int node)
{
  CRANNode *current;

  // obtain a free node
```

```
  if (!avail)
  {
    current = new CRAMNode;
  }
  else
  {
    current = avail;
    avail = avail->next;
  }


  // update node_mapper
  node_mapper[node] = current;

  current->next = temp_node;
  current->prev = NULL;

  if (temp_node)
    temp_node->prev = current;
  temp_node = current;

  return current;
}


inline void free_Node (int node)
{
  CRAMNode *current = node_mapper[node];
  if (current == NULL) return;

  node_mapper[node] = NULL;

  if (!current->next && !current->prev)
  {
    temp_node = NULL;
  }
  else
  {
    if (current->next) current->next->prev = current->prev;
    if (current->prev)
      current->prev->next = current->next;
    else
      temp_node = current->next;
  }
  current->next = avail;
  current->prev = NULL;

  avail = current;

}


// need to store a value into some node,
// so a CRAMNode should be created for the node if it doesn't
// already exist

cboolean *Obtain_Node_f_mapper (int node)
{
  CRAMNode *current = node_mapper[node];

  if (current == NULL)
    return NULL;
  else
    return &current->data;
}

cboolean *Obtain_Node (int node)
{
  CRAMNode *current = node_mapper[node];

  if (current == NULL)
  {
    current = alloc_Node(node);
  }

  current_gate_num = node;
  return &current->data;
}

cboolean *Obtain_Node_w_Loading (int node)
{
  CRAMNode *current = node_mapper[node];

  if (current == NULL)
  {
    current = alloc_Node(node);
    Load_Node(node, current->data);
  }

  return &current->data;
}


// the node is used,
// the node becomes available if current_gate_num == max_ref[node]

cboolean *Use_Node (int node)
{
  CRAMNode *current = node_mapper[node];

  if (current == NULL)
  {
    current = alloc_Node(node);

    Load_Node(node, current->data);
  }

  if (max_ref[node] == current_gate_num)
  {
    free_Node(node);
  }

  return &current->data;
}


//////////
//
// following is a set of gates designed to use the cram_mapper module.
// in order to insert faults during simulation, a two step gate
// evaluation process is needed.
// one for updating normal gates, and one for insert fault
//
//////////

inline void g_inpt (Circuit *cut,int  j)
{
  /* do nothing */
}


inline void g_buff (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node  = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  out_node->operate (0,op_x.grouperite);
}


inline void g_not  (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node  = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  out_node->operate (0,op_xbar.grouperite);
}


inline void g_and  (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node  = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  for (int l=1; l<cut->gate_list[j].n_fan_in; l++)
  {
    in_node = Use_Node (cut->gate_list[j].in_node[l]);
    in_node->operate (0,op_xandm.writex);
  }

  out_node->operate (0,op_x.grouperite);
}


inline void g_nand (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node  = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  for (int l=1; l<cut->gate_list[j].n_fan_in; l++)
  {
    in_node = Use_Node (cut->gate_list[j].in_node[l]);
    in_node->operate (0,op_xNop_m.writex);
  }

  out_node->operate (0,op_xbar.grouperite);
}


inline void g_or   (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  for (int l=1; l<cut->gate_list[j].n_fan_in; l++)
  {
    in_node = Use_Node (cut->gate_list[j].in_node[l]);
    in_node->operate (0,op_xorm.writex);
  }

  out_node->operate (0,op_x.grouperite);
}
```

```
inline void g_nor  (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  for (int l=1; l<cut->gate_list[j].n_fan_in; l++)
  {
    in_node = Use_Node (cut->gate_list[j].in_node[l]);
    in_node->operate (0,op_xorm.writex);
  }

  out_node->operate (0,op_xbar,groupwrite);
}


inline void g_xor  (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  for (int l=1; l<cut->gate_list[j].n_fan_in; l++)
  {
    in_node = Use_Node (cut->gate_list[j].in_node[l]);
    in_node->operate (0,op_xorm.writex);
  }

  out_node->operate (0,op_x.groupwrite);
}


inline void g_xnor (Circuit *cut,int  j)
{
  cboolean *out_node = Obtain_Node (cut->gate_list[j].out_node);

  cboolean *in_node = Use_Node (cut->gate_list[j].in_node[0]);
  in_node->operate (0,op_m.writex);

  for (int l=1; l<cut->gate_list[j].n_fan_in; l++)
  {
    in_node = Use_Node (cut->gate_list[j].in_node[l]);
    in_node->operate (0,op_xorm.writex);
  }

  out_node->operate (0,op_xbar,groupwrite);
}


void Gate_Eval (Circuit *cut,int j)
{
  switch (cut->gate_list[j].gate_type)
  {
    case g_INPT : g_inpt(cut,j); break;
    case g_BUFF : g_buff(cut,j); break;
    case g_NOT  : g_not (cut,j); break;
    case g_AND  : g_and (cut,j); break;
    case g_NAND : g_nand(cut,j); break;
    case g_OR   : g_or  (cut,j); break;
    case g_NOR  : g_nor (cut,j); break;
    case g_XOR  : g_xor (cut,j); break;
    case g_XNOR : g_xnor(cut,j); break;
  }
}

// end of cram_mapper.C
```

A.2 simf Benchmark Fault Simulator

```
/*
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*RAM Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
 * All rights reserved
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * simulate.C
 *
 * Purpose :  This file contains the routines for performing fault
 *            simulation using the PPSFP algorithm.
 *
 * Compile Options :
 *   ACCOUNT - inserts statements to obtain more information
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <ctype.h>
#include "structure.h"
#include "prototype.h"
#include "heap.h"

#include <stopwatch.h>

#define NumPV  256
//extern int NumPV;

#define pattern_set(a,b)    pattern[(a)*g_n_input + (b)]

/* accounting variables */

#ifdef ACCOUNT
  int g_triggered;
  unsigned int utilize_p;
  unsigned int used_PE;
  unsigned int g_sim_run;      // number of bad simulation run
  unsigned int g_sim_need;     // number of bad simulation needed
#endif

int   g_n_fault_gate_eval;
int   g_n_good_gate_eval;
int   g_n_total_pass;

int   g_total_pattern;
int   g_pattern_left;
int   g_pattern_sim;       // number of pattern in simulation
int   g_n_input;
char *pattern;

//#define MIN(a,b) ((a)<(b))?(a):(b)
//#define MAX(a,b) ((a)>(b))?(a):(b)

int  *good_value;
int  *wire_value;
#define GOOD_VALUE(a,b) good_value[(a)*NumPV/INT_SIZE + (b)]
#define WIRE_VALUE(a,b) wire_value[(a)*NumPV/INT_SIZE + (b)]

#define WIRE_SET(a,b)   memset (wire_value+(a)*NumPV/INT_SIZE,b,NumPV/8)
#define WIRE_CPY(a,b)   memcpy (wire_value+(a)*NumPV/INT_SIZE, \
                wire_value+(b)*NumPV/INT_SIZE, \
                NumPV/8)

int  *or_mask;
int  *and_mask;

/* the following variables are used for transition faults */
char *prev_value;     /* stores the last value of the
                         prev set of testv patterns */
char  first_time;     /* indicates first set of patterns */

/* the following global variables are made global to enhance speed */
Circuit *g_cut;
char  *event_list;    /* if a gate is affected by a fault, the
                         event_list of that gate is set */


/* funciton prototypes */
void Check_Fault (Circuit *cut, Fault *fault_head);
int  Good_Sim (Circuit *cut);
int  Bad_Sim (Circuit *cut, int  start_gate);

#ifdef ACCOUNT
int  Mark_detected (Fault *fault);
#else
void Mark_detected (Fault *fault);
#endif

/*
 * Init_Sim
 *
 * Initialize the simulation
 *
 */
void Init_Sim (Circuit *cut, int  n_pattern)
{
  #ifdef ACCOUNT
    used_PE = 0;
    utilize_p = 0;
    g_sim_run = 0;
    g_sim_need = 0;
  #endif

  first_time = 1;

  g_cut       = cut;
  g_pattern_left  = n_pattern;
  g_total_pattern = n_pattern;
  g_n_input       = cut->num_input;

  pattern = (char *) malloc (NumPV * g_n_input * sizeof (char));
  event_list  = (char *) malloc (cut->num_gate);
```

```
  memset (event_list, 0, cut->num_gate);
  prev_value  = (char *) malloc (cut->num_gate);

  good_value = (int  *) malloc (NumPV/8 * (cut->num_gate+1));
  wire_value = (int  *) malloc (NumPV/8 * (cut->num_gate+1));

  or_mask  = (int *) malloc (sizeof (int) * NumPV/INT_SIZE);
  and_mask = (int *) malloc (sizeof (int) * NumPV/INT_SIZE);

  if (!pattern || !event_list ||
      !prev_value || !good_value || !wire_value)
  {
    fprintf (stderr, "Error: unable to allocate enough memory\n");
    exit(1);
  }

  init_heap();
  g_n_fault_gate_eval = 0;
  g_n_good_gate_eval = 0;
  g_n_total_pass = 0;
}



/*
 * Read_Pattern
 *
 * Read NumPV patterns from input and store them into the
 * pattern data structure
 *
 * assumption :  assume the input pattern length is correct
 *               and does not contain any white space
 */
int  Read_Pattern (Circuit *cut)
{
  int  i;
  int  ch;
  int  count_char=0;
  int  count_pattern=0;

  /* store the previous good pattern value */
  for (i=0; i<cut->num_gate; i++)
    prev_value[i] = (char) GOOD_VALUE(i, NumPV/INT_SIZE - 1);

  if (g_total_pattern != g_pattern_left) first_time = 0;
  if (!g_pattern_left) return 0;


  // fill up the pattern storage area
  count_pattern = count_char = 0;

  int status = 0;
  while ((ch = getc(stdin)) != EOF)
  {
    if (ch == '\n')
    {
      count_pattern++;
      g_pattern_left--;
      if (!g_pattern_left) { status = 1; break;}
      if (count_pattern == NumPV ) { status = 2; break;}
      if ((count_char % g_n_input) != 0) {status = 4; break;}
      else continue;
    }
    else
    {
      if (isspace (ch)) continue;
      if (count_char == (NumPV * g_n_input)) {status = 3; break;}
      pattern[count_char++] = (ch == '1');
    }
  }

  if (status == 4 || count_pattern * g_n_input != count_char)
  {
    printf ("Error reading input patterns: %d != %d\n",
        count_pattern * g_n_input, count_char);
    printf ("    due to ");
    switch (status)
    {
      case 1: printf ("no pattern left!\n"); break;
      case 2: printf ("got enough patterns!\n"); break;
      case 3: printf ("got enough characters (abnormal)!\n"); break;
      case 4: printf ("not enough bits in line %d!\n",count_pattern);
        break;
    }
    exit (0);
  }

  /*
  NumPV/INT_SIZE  = count_pattern>>5;
  if (count_pattern & 0x1F) NumPV/INT_SIZE ++;
  */

  // set up the masks
  memset (or_mask, 0, NumPV/8);
  memset (and_mask, -1, NumPV/8);

  for (i=count_pattern;i<NumPV; i++)
  {
    /* the all one mask is an or mask */
    or_mask[i/INT_SIZE] <<= 1;
    or_mask[i/INT_SIZE] |= 1;
    /* the all one mask is an and mask */
    and_mask[i/INT_SIZE] <<= 1;
    and_mask[i/INT_SIZE] |= 0;
  }

  return count_pattern;
}



/*
 * Fault_Sim
 *
 * The main interface of this module
 * Calling this routine would cause the program to start
 * reading in test patterns and arrange the faults for simulation
 *
 * This routine implements the sequential PPSFP algorithm
 *
 */

#define PRINT_PROCESS {                 \
  int  n_sa, n_d, n_so;                 \
  Get_Count(&n_sa, &n_d, &n_so);        \
  printf ("%8d %7.2f %7.2f %7.2f\n",    \
      g_total_pattern - g_pattern_left,     \
      (g_total_sa - n_sa)*100.0/g_total_sa,   \
      (g_total_d  - n_d )*100.0/g_total_d ,   \
      (g_total_so - n_so)*100.0/g_total_so    \
      );                                \
}
int  Fault_Sim(Circuit *cut, Fault *fault_list, int  n_pattern)
{
  Init_Sim (cut, n_pattern);
  Fault *fault_head=NULL;
  Restruct_Fault_List (&fault_head, fault_list);

  printf ("%8s%8s%8s%8s\n","Pattern","SA","D","SO");
  PRINT_PROCESS;

  //ZEROCLOCK;
  while (Read_Pattern (cut))
  {
    //STARTCLOCK;
    Good_Sim (cut);

    #ifdef ACCOUNT
      g_triggered=0;
    #endif

    Check_Fault (cut, fault_head);
    //STOPCLOCK;

    #ifdef ACCOUNT
    {
      int  n_sa, n_d, n_so;
      Get_Count(&n_sa, &n_d, &n_so);

      printf ("%8d %d %d %d %d %f %d %d\n",
        g_total_pattern - g_pattern_left,
        n_sa, n_d, n_so,
        n_sa + n_d + n_so,
        (utilise_p*100.00/(used_PE*NumPV)),
        utilise_p,used_PE*NumPV);

    }
    #endif

    if (!Restruct_Fault_List (&fault_head, fault_list))
    {
      PRINT_PROCESS;
      break;
    }
    PRINT_PROCESS;

  }

  printf ("\n");
  #ifdef ACCOUNT
    printf ("pp_utilization rate = %f%% (%d/%d)\n",
        (utilise_p*100.00/(used_PE*NumPV)),
       utilise_p,used_PE*NumPV);

    printf ("number of bad sim run   = %d\n", g_sim_run);
    printf ("number of bad sim needed = %d\n", g_sim_need);
    printf ("sim reduction ratio     = %6.2f%%\n",
        (g_sim_need - g_sim_run) * 100.0 / g_sim_need);
  #endif

  printf ("total number of gate evaluation = %d (%d PV)\n",
      (g_n_fault_gate_eval + g_n_good_gate_eval) * NumPV,
      NumPV);
  printf ("total number of gates in circuit = %d\n", cut->num_gate);
  printf ("number of gate evaluation per pass = %f\n",
      g_n_fault_gate_eval + 1.0 / g_n_total_pass);
  printf ("reduction in percentage using heap = %f%%\n",
      ( (cut->num_gate -
        (g_n_fault_gate_eval + 1.0 / g_n_total_pass) ) * 100.0
        / cut->num_gate));
  //PRINTCLOCK;
}
```

```
/*
 * Mark_detected
 *
 * marks a tree of faults as detected by implication recursively
 *
 */
#ifdef ACCOUNT
int  Mark_detected (Fault *fault)
{
  int result = 1;
  fault->flag |= fd_IMPLIED;

  if (fault->implied1)
    if (fault->implied1->flag == fd_UNDETECTED)
    result += Mark_detected (fault->implied1);

  if (fault->implied2)
    if (fault->implied2->flag == fd_UNDETECTED)
    result += Mark_detected (fault->implied2);

  return result;
}
#else
void Mark_detected (Fault *fault)
{
  fault->flag |= fd_IMPLIED;

  if (fault->implied1)
    if (fault->implied1->flag == fd_UNDETECTED)
    Mark_detected (fault->implied1);

  if (fault->implied2)
    if (fault->implied2->flag == fd_UNDETECTED)
    Mark_detected (fault->implied2);

  return;
}
#endif



/*
 * Check_Fault
 *
 * Checks the triggering condition of a fault
 *
 */
void Check_Fault (Circuit *cut, Fault *fault_head)
{
  Fault *fault;
  #ifdef ACCOUNT
  int  usedbits[NumPV/INT_SIZE];
  #endif

  for (fault = fault_head; fault!=NULL; fault= fault->next)
  {

    if (fault->flag != fd_UNDETECTED) continue;

    /*
    printf ("checking fault %s(%d)\n",
      fault_table[fault->type],fault->wire_num);
    */

    char run=0;
    int  start_gate;
    int  k;

    int  fault_node = cut->wire_list[fault->wire_num].node_num;
    int  from_gate  = cut->wire_list[fault->wire_num].from_gate;
    int  to_gate    = cut->wire_list[fault->wire_num].to_gate;


    switch (fault->type)
    {
      /*********************
       * if value of wire = 1, fault is triggered
       *********************/
      case f_SA0:
      {
        int  flag=0;

        // fault triggering
        for (k=0; k<NumPV/INT_SIZE ; k++)
        {
          flag |= (GOOD_VALUE(fault_node, k) & and_mask[k]);
          #ifdef ACCOUNT
          usedbits[k] = (GOOD_VALUE(fault_node, k) & and_mask[k]);
          #endif
        }

        // setup wire value
        if (flag)
        {
          if (!cut->gate_list[from_gate].n_fan_out)
          {
            #ifdef ACCOUNT
              //g_triggered++;
            #endif
            //printf ("detected!!\n");
            fault->flag |= fd_DETECTED;
          }
          else
          {
            run=1;
            if (to_gate)
            {
              for (k=0; k<NumPV/INT_SIZE ; k++)
              {
                WIRE_VALUE(cut->num_gate,k) =
                  GOOD_VALUE(fault_node,k) & or_mask[k];
              }
            }
            else
            {
              for (k=0; k<NumPV/INT_SIZE ; k++)
              {
                WIRE_VALUE(fault_node,k) =
                  GOOD_VALUE(fault_node,k) & or_mask[k];
              }
            }
          }
        }
      }
      break;


      /*********************
       * if value of wire = 0, fault is triggered
       *********************/
      case f_SA1:
      {
        int  flag=0;

        // fault triggering
        for (k=0; k<NumPV/INT_SIZE ; k++)
        {
          flag |= ~(GOOD_VALUE(fault_node, k) | or_mask[k]);
          #ifdef ACCOUNT
          usedbits[k] = ~(GOOD_VALUE(fault_node, k) | or_mask[k]);
          #endif
        }

        // setup wire value
        if (flag)
        {
          if (!cut->gate_list[from_gate].n_fan_out)
          {
            #ifdef ACCOUNT
              //g_triggered++;
            #endif
            //printf ("detected!!\n");
            fault->flag |= fd_DETECTED;
          }
          else
          {
            run=1;
            if (to_gate)
            {
              for (k=0; k<NumPV/INT_SIZE ; k++)
              {
                WIRE_VALUE(cut->num_gate,k) =
                  GOOD_VALUE(fault_node,k) | and_mask[k];
              }
            }
            else
            {
              for (k=0; k<NumPV/INT_SIZE ; k++)
              {
                WIRE_VALUE(fault_node,k) =
                  GOOD_VALUE(fault_node,k) | and_mask[k];
              }
            }
          }
        }
      }
      break;


      /*********************
       * triggered if there is a 0->1 transition
       * an intermediate PO of OR gate is the same as SA at output
       *********************/
      case f_SA:
      case f_IPO:
      {
        int  temp;
        int  flag;
        int  transition[NumPV/INT_SIZE];


        //////
        // setup wire value for shifting
        //////

        temp = GOOD_VALUE (fault_node,-1);
        GOOD_VALUE(fault_node,-1) =
          (first_time)? ~1 : prev_value[fault_node];


        //////
        // check if fault is triggered
        //////
```

```
flag = 0;
for (k=0; k<NumPV/INT_SIZE ; k++)
{
  int  shifted;

  shifted =
    ((GOOD_VALUE(fault_node,k) >> 1) &
     ~( 1 << (INT_SIZE-1) )) |
    (GOOD_VALUE(fault_node,k-1)<<(INT_SIZE-1));

  flag |=
  (
    transition[k] =
      ( GOOD_VALUE(fault_node,k) ^ shifted) &
      GOOD_VALUE(fault_node,k) &
      and_mask[k]
  );

  #ifdef ACCOUNT
  usedbits[k] =
  (
    transition[k] =
      ( GOOD_VALUE(fault_node,k) ^ shifted) &
      GOOD_VALUE(fault_node,k) &
      and_mask[k]
  );
  #endif
}


/////
// restore wire value
/////

GOOD_VALUE (fault_node,-1) = temp;


/////
// setup wire value
/////

if (flag)
{
  if (!cut->gate_list[from_gate].n_fan_out)
  {
    #ifdef ACCOUNT
      //g_triggered++;
    #endif
    //printf ("detected!!\n");
    fault->flag |= fd_DETECTED;
  }
  else
  {
    run=1;
    if (to_gate)
    {
      for (k=0; k<NumPV/INT_SIZE ; k++)
      {
        WIRE_VALUE(cut->num_gate,k) =
          GOOD_VALUE(fault_node,k)|transition[k];
      }
    }
    else
    {
      for (k=0; k<NumPV/INT_SIZE ; k++)
      {
        WIRE_VALUE(fault_node,k) =
          GOOD_VALUE(fault_node,k)|transition[k];
      }
    }
  }
}
break;


/*********************
 * triggered if there is a 1->0 transition
 * an intermediate NO of AND gate is the same as SF at output
 *********************/
case f_SF :
case f_NO :
{
  int  temp;
  int  flag;
  int  transition[NumPV/INT_SIZE];


  /////
  // setup good wire value for shifting
  /////

  temp = GOOD_VALUE (fault_node,-1);
  GOOD_VALUE(fault_node,-1) =
    (first_time)? 0 : prev_value[fault_node];


  /////
  // check if fault is triggered
  /////

  flag = 0;
  for (k=0; k<NumPV/INT_SIZE ; k++)
  {
    int  shifted;

    shifted =
      ( (GOOD_VALUE(fault_node,k) >> 1) &
        ~( 1 << (INT_SIZE-1) )) |
      ( GOOD_VALUE(fault_node,k-1)<<(INT_SIZE-1));

    flag |=
    (
      transition[k] =
        ( GOOD_VALUE(fault_node,k) ^ shifted) &
        shifted & and_mask[k]
    );

    #ifdef ACCOUNT
    usedbits[k] =
    (
      transition[k] =
        ( GOOD_VALUE(fault_node,k) ^ shifted) &
        shifted & and_mask[k]
    );
    #endif
  }


  /////
  // restore good wire value
  /////

  GOOD_VALUE (fault_node,-1) = temp;


  /////
  // setup wire value
  /////

  if (flag)
  {
    if (!cut->gate_list[from_gate].n_fan_out)
    {
      #ifdef ACCOUNT
        //g_triggered++;
      #endif
      //printf ("detected!!\n");
      fault->flag |= fd_DETECTED;
    }
    else
    {
      run=1;
      if (to_gate)
      {
        for (k=0; k<NumPV/INT_SIZE ; k++)
        {
          WIRE_VALUE(cut->num_gate,k) =
            GOOD_VALUE(fault_node,k)|transition[k];
        }
      }
      else
      {
        for (k=0; k<NumPV/INT_SIZE ; k++)
        {
          WIRE_VALUE(fault_node,k) =
            GOOD_VALUE(fault_node,k)|transition[k];
        }
      }
    }
  }
}
break;

/*********************
 * a NO fault is triggered if there is a transition at
 * the gate output and the faulty input is 1
 *********************/
case f_NO :
{
  int  temp;
  int  flag;
  int  transition[NumPV/INT_SIZE];
  int  shifted[NumPV/INT_SIZE];
  int  gate_output;    // wire number of gate output

  gate_output = cut->gate_list[to_gate].out_node;


  /////
  // setup right-shifted pattern
  /////

  temp = GOOD_VALUE (gate_output,-1);
  GOOD_VALUE(gate_output,-1) =
    (first_time)?
      GOOD_VALUE(gate_output,0)>>(INT_SIZE-1) :
      prev_value[gate_output];

  for (k=0; k<NumPV/INT_SIZE ; k++)
```

```
{
  shifted[k] =
    ( (GOOD_VALUE(gate_output,k) >> 1) &
     ~( 1 << (INT_SIZE-1) )) |
    ( GOOD_VALUE(gate_output,k-1)<<(INT_SIZE-1));
}

GOOD_VALUE (gate_output,-1) = temp;


/////
// check if fault is triggered
/////

flag = 0;
for (k=0; k<NumPV/INT_SIZE ; k++)
{
  flag |=
  (
    transition[k] =
      ( GOOD_VALUE(gate_output,k) ^ shifted[k]) &
      GOOD_VALUE(fault_node,k) &
      and_mask[k]
  );

  #ifdef ACCOUNT
  usedbits[k] =
  (
    transition[k] =
      ( GOOD_VALUE(gate_output,k) ^ shifted[k]) &
      GOOD_VALUE(fault_node,k) &
      and_mask[k]
  );
  #endif
}


/////
// setup wire value
/////

if (flag)
{
  if (!cut->gate_list[from_gate].n_fan_out)
  {
    #ifdef ACCOUNT
      //g_triggered++;
    #endif
    //printf ("detected!!\n");
    fault->flag |= fd_DETECTED;
  }
  else
  {
    run=1;
    if (to_gate)
    {
      for (k=0; k<NumPV/INT_SIZE ; k++)
      {
        WIRE_VALUE(cut->num_gate,k) =
          GOOD_VALUE(fault_node,k)&~(transition[k]);
      }
    }
    else
    {
      for (k=0; k<NumPV/INT_SIZE ; k++)
      {
        WIRE_VALUE(fault_node,k) =
          GOOD_VALUE(fault_node,k)&~(transition[k]);
      }
    }
  }
}
break;


/*********************
 * a PO fault is triggered if there is a transition at
 * the gate output and the faulty input is 0.
 *********************/
case f_PO :
{
  int  temp;
  int  flag;
  int  transition[NumPV/INT_SIZE];
  int  shifted[NumPV/INT_SIZE];
  int  gate_output;    // wire number of gate output

  gate_output = cut->gate_list[to_gate].out_node;


  /////
  // setup gate_output for shifting
  /////

  temp = GOOD_VALUE (gate_output,-1);
  GOOD_VALUE(gate_output,-1) =
    (first_time)?
      GOOD_VALUE(gate_output,0)>>(INT_SIZE-1) :
      prev_value[gate_output];
```

```
  for (k=0; k<NumPV/INT_SIZE ; k++)
  {
    shifted[k] =
      ((GOOD_VALUE(gate_output,k) >> 1) &
      ~( 1 << (INT_SIZE-1) )) |
      (GOOD_VALUE(gate_output,k-1)<<(INT_SIZE-1));
  }

  GOOD_VALUE (gate_output,-1) = temp;


  /////
  // check if fault is triggered
  /////

  flag = 0;
  for (k=0; k<NumPV/INT_SIZE ; k++)
  {
    flag |=
    (
      transition[k] =
        (GOOD_VALUE(gate_output,k) ^ shifted[k]) &
        ~(GOOD_VALUE(fault_node,k)) &
        and_mask[k]
    );

    #ifdef ACCOUNT
    usedbits[k] =
    (
      transition[k] =
        (GOOD_VALUE(gate_output,k) ^ shifted[k]) &
        ~(GOOD_VALUE(fault_node,k)) &
        and_mask[k]
    );
    #endif
  }


  /////
  // setup wire value
  /////

  if (flag)
  {
    if (!cut->gate_list[from_gate].n_fan_out)
    {
      #ifdef ACCOUNT
        //g_triggered++;
      #endif
      //printf ("detected!!\n");
      fault->flag |= fd_DETECTED;
    }
    else
    {
      run=1;
      if (to_gate)
      {
        for (k=0; k<NumPV/INT_SIZE ; k++)
        {
          WIRE_VALUE(cut->num_gate,k) =
            GOOD_VALUE(fault_node,k)|transition[k];
        }
      }
      else
      {
        for (k=0; k<NumPV/INT_SIZE ; k++)
        {
          WIRE_VALUE(fault_node,k) =
            GOOD_VALUE(fault_node,k)|transition[k];
        }
      }
    }
  }
  break;

  default: break;

}


if (run)
{
#ifdef ACCOUNT
  //g_triggered++;
#endif

  new_heap();

  /////
  // perpare fan_out wires, start_gate
  /////

  int  i;

  if (!to_gate)
  {

    /////
    // wire has fan_out
    /////
```

```
      start_gate = cut->wire_list
            [cut->gate_list[from_gate].fan_out[0]].to_gate;
      event_list[start_gate] = 1;
      add_heap(start_gate);

      for (i=1; i<cut->gate_list[from_gate].n_fan_out; i++)
      {
        int  temp = cut->wire_list[cut->gate_list[from_gate]
                   .fan_out[i]].to_gate;
        start_gate = MIN (start_gate, temp);
        event_list[temp] = 1;
        add_heap(temp);
      }
    }
    else
    {

      /////
      // wire has no fan_out
      /////

      start_gate = to_gate;
      event_list[start_gate] = 1;
      add_heap(start_gate);


      /////
      // change the reference to the node
      /////

      for (i=0; i<cut->gate_list[to_gate].n_fan_in; i++)
      {
        if (cut->gate_list[to_gate].in_node[i] == fault_node)
        {
          cut->gate_list[to_gate].in_node[i] =
            cut->num_gate;
          break;
        }
      }
    }



    /////
    // Simulation
    /////


    int util;
    if (util = Bad_Sim (cut, start_gate))
    {
      //printf ("detected!!\n");
    #ifdef ACCOUNT
      g_sim_need += Mark_detected(fault) - 1;
    #else
      Mark_detected(fault);
    #endif

      fault->flag = fd_DETECTED;
    }

    // Reset wire_value array
    // the value of i is result from the perparation section

    if (to_gate) cut->gate_list[to_gate].in_node[i] = fault_node;

    memcpy (wire_value + fault_node * NumPV/INT_SIZE,
        good_value + fault_node * NumPV/INT_SIZE,
        NumPV/8);


    #ifdef ACCOUNT

    g_sim_run++;
    g_sim_need++;

    //
    // accumulate the number of PEs used for useful simulation
    //
    if (!util) util = NumPV;
    for (int ak=1; ak<=util; ak++)
    {
      int temp = usedbits[(ak-1)/INT_SIZE];
      int shift = ak%INT_SIZE;
      if (!shift) shift = INT_SIZE;
      temp >>= (INT_SIZE - shift);
      if (temp & 1) utilize_p++;
    }
    used_PE++;

    #endif
  }
 }
 return;
}


inline void g_inpt (int  j)
{
  int  k;

  /* load input pattern */
  for (k=0; k<NumPV; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k/INT_SIZE) <<=1;
    WIRE_VALUE(g_cut->gate_list[j].out_node,k/INT_SIZE) |=
      pattern_set(k,g_cut->gate_list[j].input_num) & 1;
  }
}

inline void g_buff (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);
}

inline void g_not  (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      ~WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);
}

inline void g_and  (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
      WIRE_VALUE(g_cut->gate_list[j].out_node,k) &=
        WIRE_VALUE(g_cut->gate_list[j].in_node[l],k);
  }
}

inline void g_nand (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
      WIRE_VALUE(g_cut->gate_list[j].out_node,k) &=
        WIRE_VALUE(g_cut->gate_list[j].in_node[l],k);

    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      ~WIRE_VALUE(g_cut->gate_list[j].out_node,k);
  }
}


inline void g_or   (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
      WIRE_VALUE(g_cut->gate_list[j].out_node,k) |=
        WIRE_VALUE(g_cut->gate_list[j].in_node[l],k);
  }
}

inline void g_nor  (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
      WIRE_VALUE(g_cut->gate_list[j].out_node,k) |=
        WIRE_VALUE(g_cut->gate_list[j].in_node[l],k);

    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      ~WIRE_VALUE(g_cut->gate_list[j].out_node,k);
  }
}

inline void g_xor  (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
      WIRE_VALUE(g_cut->gate_list[j].out_node,k) ^=
        WIRE_VALUE(g_cut->gate_list[j].in_node[l],k);
  }
}

inline void g_xnor (int  j)
{
  for (int k=0; k<NumPV/INT_SIZE ; k++)
  {
    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      WIRE_VALUE(g_cut->gate_list[j].in_node[0],k);

    for (int l=1; l<g_cut->gate_list[j].n_fan_in; l++)
```

```
      WIRE_VALUE(g_cut->gate_list[j].out_node,k) ^=
        WIRE_VALUE(g_cut->gate_list[j].in_node[1],k);

    WIRE_VALUE(g_cut->gate_list[j].out_node,k) =
      ~WIRE_VALUE(g_cut->gate_list[j].out_node,k);
  }
}


inline void Gate_Map (Circuit *cut, int j)
{
  switch (cut->gate_list[j].gate_type)
  {
    case g_INPT : g_inpt(j); break;
    case g_BUFF : g_buff(j); break;
    case g_NOT  : g_not (j); break;
    case g_AND  : g_and (j); break;
    case g_NAND : g_nand(j); break;
    case g_OR   : g_or  (j); break;
    case g_NOR  : g_nor (j); break;
    case g_XOR  : g_xor (j); break;
    case g_XNOR : g_xnor(j); break;
  }

}


/*
 * Bad_Sim
 *
 * start_gate - the gate to start from
 *
 * preconditions:
 *  for delay faults, since all the fan_out of a wire will have the
 *  same fault values, these wires should all be set to the faulty
 *  value before this simulation routine is called.
 *
 * return the gate that detectes the fault.0 if fault is not detected
 *
 */
int  Bad_Sim (Circuit *cut, int  start_gate)
{
  int  i,j,k,l;    /* for loop counters */
  int  detected = 0;

  int  dirty_node[cut->num_gate];
  int  event_done = 0;

  g_n_total_pass++;

  while ((j = pop_heap()) >= 0)
  {
    dirty_node[event_done++]=j;
    event_list[j] = 0;


    /////
    // simulate
    /////

    Gate_Map (cut, j);
    g_n_fault_gate_eval++;


    /////
    // check for changes in output value
    /////

    int  flag = 0;

    for (k=0;k<NumPV/INT_SIZE ; k++)
    {
      flag |= WIRE_VALUE(cut->gate_list[j].out_node,k) ^
          GOOD_VALUE(cut->gate_list[j].out_node,k) ;
    }

    if (flag)
    {
      if (cut->gate_list[j].n_fan_out)
      {
        /////
        // add event for further propagation
        /////

        for (i=0;i<cut->gate_list[j].n_fan_out; i++)
        {
          int  nexte = cut->wire_list[
            cut->gate_list[j].fan_out[i] ].to_gate;

          if (!event_list[nexte])
          {
            event_list[nexte] = 1;
            add_heap(nexte);
          }
        }
      }
      else
      {
        /////
        // primary output
        /////

#ifdef ACCOUNT
        //
        // find the first PE that detects the fault
        // PE number starts with 1 in this case
        //
        int temp;
        for (k=0; k<NumPV/INT_SIZE; k++)
        {
          if (temp =
            (WIRE_VALUE(cut->gate_list[j].out_node,k)^
             GOOD_VALUE(cut->gate_list[j].out_node,k)))break;
        }
        for (l=1; l<=INT_SIZE; l++)
          if ((temp >> (INT_SIZE-l)) & 1) break;

        if (!detected)
          detected = k*INT_SIZE + l;
        else
        {
          detected = (k*INT_SIZE + l) < detected ?
                (k*INT_SIZE + l) : detected ;

        }

        /*
        for (int ak=0; ak<NumPV/INT_SIZE; ak++)
        {
          printf
          ("%16x ",GOOD_VALUE(cut->gate_list[j].out_node,ak) ^
              WIRE_VALUE(cut->gate_list[j].out_node,ak) );
        }
        printf ("\n");
        printf ("  j = %d, k = %d, l = %d => detected = %d\n",
              j, k, l, detected);
        */
#else
        detected = j;
        goto end_Bad_Sim;
#endif
      }
    }
  }

  end_Bad_Sim:


  while (j=read_heap()) event_list[j] = 0;

  // clean the wire_value for next time
  for (j=0; j<event_done; j++)
  {
    memcpy (wire_value + dirty_node[j] * NumPV/INT_SIZE,
        good_value + dirty_node[j] * NumPV/INT_SIZE,
        NumPV/8);
  }


  return detected;

}


/*
 * Good_Sim
 *
 * preconditions:
 *  for delay faults, since all the fan_out of a wire will have the
 *  same fault values, these wires should all be set to the faulty
 *  value before this simulation routine is called.
 *
 * if faulty sim - return 1 if fault is detected,
 *            0 if fault is not detected
 *
 */
int  Good_Sim (Circuit *cut)
{
  int  i,j,k,l;    /* for loop counters */

  /* start going thru the circuit */
  for (j=0; j<cut->num_gate; j++)
  {
    g_n_good_gate_eval++;
    Gate_Map (cut, j);
  }

  memcpy (good_value, wire_value, NumPV/8 * cut->num_gate);

  return 0;

}

// end of simulate.C
```

A.3 Pattern-Parallel Fault Simulator

```
/*
```

```
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
* Parallel Fault Simulators on the C*RAN Architecture
*
* Copyright (c) 1997, 1998 by Albert L -C  Kwong, Edmonton, Canada
* All rights reserved.
* This software may be used for non-profit university research
* if given the author's expressed permission.  An executed license
* agreement with the author is required for all other uses of
* this software.  Redistribution of this software is not
* permitted without the author's expressed permission.
* This copyright notice must remain intact.
* Derivative works may contain additional notices.
*
* This software comes with no warranty.
*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*
* simulate_pps.C
*
* Purpose : This file contains the routines for performing
*           pattern-parallel fault simulation.
*
* Compile Options :
*   PEUTIL - measure PE utilization ratio
*/

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <ctype.h>
#include <cran.h>
#include <stopwatch.h>

#include "structure.h"
#include "prototype.h"
#include "heap.h"

#define op_xbarandm  0x04
#define op_xandmbar  0x50

long n_gate_eval;

#ifdef PEUTIL
int useful_PE;
int used_PE;
cboolean count_PE;
double util_time;
double stop_time;

#define STARTUTIL stop_time  = count_PE time;
#define ENDUTIL   util_time += count_PE time - stop_time;
#endif

int   g_total_pattern;
int   g_pattern_left;
int   g_num_int ;
int   g_pattern_sim;      // number of pattern in simulation
int   g_n_input;

cint *pattern;
cint *good_value;
cint *wire_value;

cboolean or_mask;

/* the following variables are used for transition faults */
char  first_time;       /* indicates first set of patterns */

/* the following global variables are made global to enhance speed */
Circuit *g_cut;
char  *event_list;      /* if a gate is affected by a fault, the
                           event_list of that gate is set */


/* funciton prototypes */
void Check_Fault (Circuit *cut, Fault *fault_head);
void Mark_detected (Fault *fault);
int  Good_Sim (Circuit *cut);
int  Bad_Sim (Circuit *cut, int  start_gate);
void cintcpy (cint &a, cint &b, int offset);


// assuming a.bits = b.bits, cpy from b to a
inline void cintcpy (cint &a, cint &b, int offset)
{
  int i;
  for (i=offset; i< a.bits; i++)
  {
    b.operate (i, op_x, writex);
    a.operate (i, op_x, groupwrite);
  }
}


/*
 * Init_Sim
 *
 * Initialize the simulation
 *
 */
void Init_Sim (Circuit *cut, int  n_pattern)
{
  first_time = 1;

  n_gate_eval = 0;

  g_cut       = cut;
  g_pattern_left  = n_pattern;
  g_total_pattern = n_pattern;
  g_n_input       = cut->num_input;

  event_list  = (char *) malloc (cut->num_gate);
  memset (event_list, 0, cut->num_gate);

  if (!event_list)
  {
    fprintf (stderr,"cannot allocate memory for event_list\n");
    exit (1);
  }

  pattern  = new cint (g_n_input, ALIGNED);
  good_value = new cint (cut->num_gate+1, ALIGNED);
  wire_value = new cint (cut->num_gate+1, ALIGNED);

  ini_heap();

  #ifdef PEUTIL
  useful_PE = used_PE = 0;
  util_time = 0;
  #endif
}



/*
 * Read_Pattern
 *
 * Read NUM_PE patterns from input and store them into the
 * pattern data structure
 *
 * assumption : assume the input pattern length is correct and does not
 *              contain any white space
 */
int  Read_Pattern (Circuit *cut)
{
  int  i;
  int  ch;
  int  count_char=0;
  int  count_pattern=0;

  if (g_total_pattern != g_pattern_left) first_time = 0;
  if (!g_pattern_left) return 0;


  // copy last PE
  (*pattern)[0] = (*pattern)[NUM_PE-1];

  count_pattern = count_char = 0;

  // setup y as mask
  pattern->operate(0, op_one, writey);
  pattern->operate(0, op_y, shiftright);


  // fill up the pattern storage area
  while ((ch = getc(stdin)) != EOF)
  {
    if (ch == '\n')
    {
      //printf ("count_pattern %d\n",count_pattern);

      count_pattern++;
      g_pattern_left--;
      pattern->operate (0, op_y, shiftright);

      if (!g_pattern_left) break;

      // we only have so many PEs
      if (count_pattern == (NUM_PE - 1)) break;

      else continue;
    }
    else
    {
      if (isspace (ch)) continue;
      if (count_char >= (NUM_PE * g_n_input)) break;

      if (ch == '1') pattern->operate (0, op_one, writex);
      else           pattern->operate (0, op_zero, writex);
      pattern->operate (count_char++%g_n_input, 0xa2, groupwrite);
    }
  }

  if ( count_pattern * g_n_input != count_char)
  {
    printf ("Error reading input patterns\n");
    exit (0);
  }


  // set up the masks
  or_mask.operate (0, op_ybar, shiftright);  // write y
```

```
  or_mask.operate (0, op_y, groupwrite);   // y mbar

  // double first pattern
  if (first_time) (*pattern)[0] = (*pattern)[1];

  g_pattern_sim = count_pattern;
  return count_pattern;
}

/*
 * Mark_detected
 *
 * marks a tree of faults as detected by implication recursively
 *
 */
void Mark_detected (Fault *fault)
{
  fault->flag |= fd_IMPLIED;

  if (fault->implied1)
    if (fault->implied1->flag == fd_UNDETECTED)
    Mark_detected (fault->implied1);

  if (fault->implied2)
    if (fault->implied2->flag == fd_UNDETECTED)
    Mark_detected (fault->implied2);

  return;
}


/*
 * Check_Fault
 *
 * Checks the triggering condition of a fault
 *
 */
void Check_Fault (Circuit *cut, Fault *fault_head)
{
  Fault *fault;

  for (fault = fault_head; fault!=NULL; fault= fault->next)
  {
/*
    printf ("checking fault %s(%d)\n",
      fault_table[fault->type],fault->wire_num);
*/
    if (fault->flag != fd_UNDETECTED) continue;

    char run=0;
    int  start_gate;
    int  k;

    int  fault_node = cut->wire_list[fault->wire_num].node_num;
    int  from_gate  = cut->wire_list[fault->wire_num].from_gate;
    int  to_gate    = cut->wire_list[fault->wire_num].to_gate;


    switch (fault->type)
    {
      /*********************
       * if value of wire = 1, fault is triggered
       *********************/
      case f_SA0:
      {
        boolean flag=true;

        // fault triggering
        or_mask.operate (0, op_m, writex);
        good_value->operate
          (fault_node, op_xnandm, busaccess,flag);

        #ifdef PEUTIL
        STARTUTIL;
        good_value->operate (fault_node, op_m, writex);
        count_PE.operate (0, op_x, groupwrite);
        ENDUTIL;
        #endif

        // setup wire value
        if (!flag)
        {
          if (!cut->gate_list[from_gate].n_fan_out)
          {
            fault->flag |= fd_DETECTED;
          }
          else
          {
            run=1;
            if (to_gate) wire_value->operate
                (cut->num_gate, op_zero, groupwrite);
            else     wire_value->operate
                (fault_node, op_zero, groupwrite);
          }
        }
      }
      break;

/*********************
 * if value of wire = 0, fault is triggered
 *********************/
case f_SA1:
{
  boolean flag = true;

  // fault triggering
  or_mask.operate (0, op_mbar, writex);
  good_value->operate
    (fault_node, op_xorm, busaccess, flag);

  #ifdef PEUTIL
  STARTUTIL;
  good_value->operate (fault_node, op_mbar, writex);
  count_PE.operate (0, op_x, groupwrite);
  ENDUTIL;
  #endif

  // setup wire value
  if (!flag)
  {
    if (!cut->gate_list[from_gate].n_fan_out)
    {
      fault->flag |= fd_DETECTED;
    }
    else
    {
      run=1;
      if (to_gate) wire_value->operate
          (cut->num_gate, op_one, groupwrite);
      else     wire_value->operate
          (fault_node, op_one, groupwrite);
    }
  }
}
break;


/*********************
 * triggered if there is a 0->1 transition
 * an intermediate PO of OR gate is the same as SR at output
 *********************/
case f_SR:
case f_iPO:
{
  /////
  // check if fault is triggered
  /////

  or_mask.operate (0, op_m, writex);

  boolean flag = true;
  good_value->operate (fault_node, op_m, shiftright);
  good_value->operate (fault_node, 0x20, writex); // x ybar m
  good_value->operate (fault_node, op_xbar, busaccess, flag);

  // x = transition mask

  #ifdef PEUTIL
  STARTUTIL;
  count_PE.operate (0, op_x, groupwrite);
  ENDUTIL;
  #endif

  /////
  // setup wire value
  /////

  if (!flag)
  {
    if (!cut->gate_list[from_gate].n_fan_out)
    {
      fault->flag |= fd_DETECTED;
    }
    else
    {
      run=1;
      int  target = (to_gate) * cut->num_gate + fault_node;

      good_value->operate (fault_node, op_xbarandm, writex);
      wire_value->operate (target, op_x, groupwrite);
    }
  }
}
break;


/*********************
 * triggered if there is a 1->0 transition
 * an intermediate PO of AND gate is the same as SF at output
 *********************/
case f_SF:
case f_iPO:
{
  /////
  // check if fault is triggered
  /////
```

```
or_mask operate (0, op_m, writex);

boolean flag = true;
good_value->operate (fault_node, op_m, shiftright);
good_value->operate (fault_node, 0x40, writex); // x y mbar
good_value->operate (fault_node, op_xbar, busaccess, flag);

// x = transition mask

#ifdef PEUTIL
STARTUTIL;
count_PE operate (0, op_x, groupwrite);
ENDUTIL;
#endif


/////
// setup wire value
/////

if (!flag)
{
  if (!cut->gate_list[from_gate].n_fan_out)
  {
    fault->flag |= fd_DETECTED;
  }
  else
  {
    run=1;
    int  target = (to_gate) ? cut->num_gate : fault_node;

    good_value->operate (fault_node, op_norm, writex);
    wire_value->operate (target, op_x, groupwrite );
  }
 }
}
break; /


/*********************
 * a NO fault is triggered if there is a transition at
 * the gate output and the faulty input is 1
 *********************/
case f_NO :
{
  /////
  // check if fault is triggered
  /////

  int  gate_output = cut->gate_list[to_gate].out_node;


  or_mask operate (0, op_m, writex);

  boolean flag = true;
  good_value->operate (gate_output, op_m, shiftright);
  good_value->operate (gate_output, 0x60, writex); // x&(y^m)
  good_value->operate (fault_node, op_xandm, writex);
  good_value->operate (fault_node, op_xbar, busaccess, flag);

  // x = transition mask

  #ifdef PEUTIL
  STARTUTIL;
  count_PE operate (0, op_x, groupwrite);
  ENDUTIL;
  #endif


  /////
  // setup wire value
  /////

  if (!flag)
  {
    if (!cut->gate_list[from_gate].n_fan_out)
    {
      fault->flag |= fd_DETECTED;
    }
    else
    {
      run=1;
      int  target = (to_gate) ? cut->num_gate : fault_node;

      good_value->operate (fault_node, op_xbarandm, writex);
      wire_value->operate (target, op_x, groupwrite );
    }
  }
}
break;


/*********************
 * a PO fault is triggered if there is a transition at
 * the gate output and the faulty input is 0
 *********************/
case f_PO
{
  int  gate_output = cut->gate_list[to_gate].out_node;


  or_mask operate (0, op_m, writex);

  boolean flag = true;
  good_value->operate (gate_output, op_m, shiftright);
  good_value->operate
    (gate_output, op_xm(op_y^op_m), writex);
  good_value->operate (fault_node, op_xandmbar, writex);
  good_value->operate
    (fault_node, op_xbar, busaccess, flag);

  // x = transition mask

  #ifdef PEUTIL
  STARTUTIL;
  count_PE operate (0, op_x, groupwrite);
  ENDUTIL;
  #endif


  /////
  // setup wire value
  /////

  if (!flag)
  {
    if (!cut->gate_list[from_gate].n_fan_out)
    {
      fault->flag |= fd_DETECTED;
    }
    else
    {
      run=1;
      int  target = (to_gate)?cut->num_gate:fault_node;

      good_value->operate (fault_node, op_norm, writex);
      wire_value->operate (target, op_x, groupwrite );
    }
  }
}
break;

default: break;
}

if (run)
{

  new_heap();

  /////
  // porpare fan_out wires, start_gate
  /////

  int  i;

  if (!to_gate)
  {

    /////
    // wire has fan_out
    /////

    start_gate = cut->wire_list
          [cut->gate_list[from_gate].fan_out[0]].to_gate;
    event_list[start_gate] = 1;
    add_heap(start_gate);

    for (i=1; i<cut->gate_list[from_gate].n_fan_out; i++)
    {
      int  temp = cut->wire_list[cut->gate_list[from_gate]
                  .fan_out[i]].to_gate;
      start_gate = MIN (start_gate, temp);
      event_list[temp] = 1;
      add_heap(temp);
    }
  }
  else
  {

    /////
    // wire has no fan_out
    /////

    start_gate = to_gate;
    event_list[start_gate] = 1;
    add_heap(start_gate);


    /////
    // change the reference to the node
    /////

    for (i=0; i<cut->gate_list[to_gate].n_fan_in; i++)
    {
      if (cut->gate_list[to_gate].in_node[i] == fault_node)
      {
        cut->gate_list[to_gate].in_node[i] = cut->num_gate;
        break;
      }
    }
  }
```

```
      /////
      // Simulation
      /////

      if (Bad_Sim (cut, start_gate))
      {
        Mark_detected(fault);
        fault->flag = fd_DETECTED;
      }

      // Reset wire_value array
      // the value of i is result from the perparation section

      if (to_gate) cut->gate_list[to_gate].in_node[i] = fault_node;

      good_value->operate (fault_node, op_m, writex);
      wire_value->operate (fault_node, op_x, groupwrite);
    }
  }
  return;
}



inline void g_inpt (int  j)
{
  /* load input pattern */
  pattern->operate  (g_cut->gate_list[j].input_num,op_m,writex);
  wire_value->operate (g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_buff (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);
  wire_value->operate (g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_not  (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);
  wire_value->operate (g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}

inline void g_and  (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);

  int  i;
  for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[i],op_xandm,writex);

  wire_value->operate (g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_nand (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);

  int  i;
  for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[i],op_xandm,writex);

  wire_value->operate (g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}


inline void g_or   (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);

  int  i;
  for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[i],op_xorm,writex);

  wire_value->operate (g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_nor  (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);

  int  i;
  for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[i],op_xorm,writex);

  wire_value->operate (g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}

inline void g_xor  (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);

  int  i;
  for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[i],op_xorm,writex);

  wire_value->operate (g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_xnor (int  j)
{
  wire_value->operate (g_cut->gate_list[j].in_node[0],op_m,writex);

  int  i;
  for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[i],op_xorm,writex);

  wire_value->operate (g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}

inline void Gate_Map (Circuit *cut, int j)
{
  switch (cut->gate_list[j].gate_type)
  {
    case g_INPT: g_inpt(j); break;
    case g_BUFF: g_buff(j); break;
    case g_NOT : g_not (j); break;
    case g_AND : g_and (j); break;
    case g_NAND: g_nand(j); break;
    case g_OR  : g_or  (j); break;
    case g_NOR : g_nor (j); break;
    case g_XOR : g_xor (j); break;
    case g_XNOR: g_xnor(j); break;
  }
}

/*
 * Bad_Sim
 *
 * start_gate - the gate to start from
 *
 * preconditions:
 *  for delay faults, since all the fan_out of a wire will have the
 *  same fault values, these wires should all be set to the faulty
 *  value before this simulation routine is called
 *
 * return the gate that detectes the fault;0 if fault is not detected
 *
 */
int  Bad_Sim (Circuit *cut, int  start_gate)
{
  int  i,j;    /* for loop counters */
  int  detected = 0;

  #ifdef PEUTIL
  STARTUTIL;
  boolean detect_PE;
  detect_PE.operate (0, op_zero, groupwrite);
  ENDUTIL;
  #endif

  int  dirty_node[cut->num_gate];
  int  event_done = 0;

  while ((j = pop_heap()) >= 0)
  {
    dirty_node[event_done++]=j;
    event_list[j] = 0;


    /////
    // simulate
    /////

    n_gate_eval++;
    Gate_Map (cut, j);

    /////
    // check for changes in output value
    /////

    boolean flag = true;

    wire_value->operate (cut->gate_list[j].out_node, op_m, writex);
    good_value->operate (cut->gate_list[j].out_node, op_x^op_m, writex);
    or_mask.operate   (0, ~(op_x & op_m), busaccess, flag);

    if (!flag)
    {
      if (cut->gate_list[j].n_fan_out)
      {
        /////
        // add event for further propagation
        /////

        for (i=0;i<cut->gate_list[j].n_fan_out; i++)
        {
          int  nextg = cut->wire_list[
            cut->gate_list[j].fan_out[i] ].to_gate;

          if (!event_list[nextg])
          {
            event_list[nextg] = 1;
            add_heap(nextg);
          }
        }
```

```
      }
      else
      {
        /////
        // primary output
        /////

        #ifdef PEUTIL
        STARTUTIL;
        if (!detected) detected = j;
        detect_PE.operate (0, op_x | op_m, groupwrite);
        ENDUTIL;
        #else
        detected = j;
        goto end_Bad_Sim;
        #endif
      }
    }
  }

end_Bad_Sim:

#ifdef PEUTIL
STARTUTIL;
if (detected)
{
  // search for the first PE that detects the fault
  // pre : detect_PE contains the PEs that detected fault
  // post: temp2 will have the bits before this PE set to 1

  cboolean temp2;
  cboolean wmask;
  detect_PE.operate (0, op_m, writey);

  temp2.operate (0, op_zero, groupwrite);
  wmask.operate (0, op_one, groupwrite);

  for (int k=(PEid.bits-1); k>=0; k--)
  {
    boolean flag = true;
    PEid.operate (k, op_m, writex);
    PEid.operate (k, op_ybar | op_x, busaccess, flag);

    if (!flag)
    {
      // in lower half
      wmask.operate (0, op_xbar & op_m, groupwrite);
      temp2.operate (0, op_y & op_xbar, writey);
    }
    else
    {
      wmask.operate (0, op_xbar & op_m, writex);
      temp2.operate (0, op_m | op_x, groupwrite);
    }
  }
  temp2.operate (0, op_m, shiftright);  // include detection
  count_PE.operate (0, op_y & op_m, writex); // triggerings
  or_mask.operate (0, op_m & op_x, writex);
  count_PE.operate (0, op_x, groupwrite);
}
else
{
  or_mask.operate (0, op_m, writex);
  count_PE.operate (0, op_m & op_x, groupwrite);
}

used_PE++;
useful_PE += count_PE.NumberOfOnes(0);
ENDUTIL;
#endif

while (j=read_heap()) event_list[j] = 0;

// clean the wire_value for next time
for (j=0; j<event_done; j++)
{
  good_value->operate (dirty_node[j], op_m, writex);
  wire_value->operate (dirty_node[j], op_x, groupwrite);
}

return detected;

}


/*
 * Good_Sim
 *
 * preconditions:
 *  for delay faults, since all the fan_out of a wire will have the
 *  same fault values, those wires should all be set to the faulty
 *  value before this simulation routine is called.
 *
 * if faulty sim - return 1 if fault is detected,
 *           0 if fault is not detected
 *
 */
int  Good_Sim (Circuit *cut)
{
  int  i,j,k,l;    /* for loop counters */

  /* start going thru the circuit */

  for (j=0; j<cut->num_gate; j++)
  {
    n_gate_eval++;
    Gate_Map (cut, j);
  }

  cintcpy(*good_value, *wire_value, 0);
  return 0;

}



/*
 * Fault_Sim
 *
 * The main interface of this module.
 * Calling this routine would cause the program to start reading in test
 * patterns and arrange the faults for simulation.
 *
 * This routine implements the sequential PPSFP algorithm.
 *
 */

#define PRINT_PROCESS {                \
  int  n_sa, n_d, n_so;                \
  Count_Fault_List(fault_list,&n_sa, &n_d, &n_so);   \
  printf ("%8d %7.2f %7.2f %7.2f\n",          \
       g_total_pattern - g_pattern_left,      \
       (g_total_sa - n_sa)*100.0/g_total_sa,  \
       (g_total_d  - n_d )*100.0/g_total_d ,  \
       (g_total_so - n_so)*100.0/g_total_so   \
       );                                     \
}


int  Fault_Sim(Circuit *cut, Fault *fault_list, int  n_pattern)
{
  Init_Sim (cut, n_pattern);
  Fault *fault_head=NULL;
  Restruct_Fault_List (&fault_head, fault_list);

  #ifndef PEUTIL
  printf ("%8s%8s%8s%8s\n","Pattern","SA","GD","SO");
  PRINT_PROCESS;
  #else
  printf ("%8s%16s%8s\n","Pattern","time","util.");
  printf ("%8d %15.9f %7.2f\n",0.0,0);
  #endif

  STOPCLOCK;

  while (Read_Pattern (cut))
  {
    Good_Sim (cut);

    Check_Fault (cut, fault_head);

    STARTCLOCK;
    #ifndef PEUTIL
    PRINT_PROCESS;
    #else
    printf ("%8d %15.9f %f\n",
       g_total_pattern - g_pattern_left,
       (count_PE.time - util_time)/1000000000.,
       useful_PE * 100.0 / (used_PE * perfParams::processors) );
    #endif;

    bool nonempty = Restruct_Fault_List (&fault_head, fault_list);
    STOPCLOCK;

    if (!nonempty) break;
  }

  #ifdef PEUTIL
  printf ("PE utilization = %6.2f (%d/%d)\n",
    useful_PE * 100.0 / (used_PE * perfParams::processors) ,
    useful_PE, used_PE * perfParams::processors);
  #endif

  printf ("number of gate eval = %d (%d PE)\n",
      n_gate_eval * NUM_PE, NUM_PE);

}

// end of simulate_pps.C
```

A.4 Fault-Parallel Fault Simulator

```
/*
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the CoRAM Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Kwong, Edmonton, Canada
```

```
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * simulate_fps.C
 *
 * Purpose : This file contains the routines for performing
 *           fault-parallel fault simulation.
 *
 * Compile Options :
 *   PEUTIL - measure PE utilization ratio
 *   FPSDFS - uses the depth first search algorithm to group faults
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <ctype.h>
#include <cram.h>
#include <stopwatch.h>

#include "structure.h"
#include "prototype.h"
#include "cram_mapper.h"
#include "heap.h"

#define op_yandnbar   op_y & op_nbar
#define op_xbarandm   op_xbar & op_m
#define op_xbarorm    op_xbar | op_m

char  first_time;       /* indicates first set of patterns */
int   n_gate_eval;      /* number of gate evaluation */

int   g_total_pattern;
int   g_pattern_left;
int   g_pattern_sim;      // number of pattern in simulation
int   g_n_input;

#ifdef PEUTIL
int useful_PE;
int used_PE;
#endif

boolean fault_free;

int  *node_value;       // int array for storing fault-free info
char *event_list;       // mark which gate is/was in heap


/*
 * Init_Sim
 *
 * Initialize the simulation
 *
 */
void Init_Sim (Circuit *cut, int  n_pattern)
{
  first_time = 1;

  n_gate_eval = 0;

  g_pattern_left  = n_pattern;
  g_total_pattern = n_pattern;
  g_n_input       = cut->num_input;

  node_value = (int *) malloc (sizeof(int) * cut->num_gate);

  event_list = (char *) malloc (cut->num_gate);
  memset (event_list, 0, cut->num_gate);

  ini_heap();
  Init_CRAM_Mapper (cut, node_value);

  fault_free.operate (0, op_one, shiftright);       // y = 0111...
  fault_free.operate (0, op_ybar, writey);          // y = 1000...
  fault_free.operate (0, op_y, groupwrite);

  #ifdef PEUTIL
  useful_PE = used_PE = 0;
  #endif

}


/*
 * Read_Pattern
 *
 * Read one pattern from input and store it in the node_value array
 *
 * assumption : assume the input pattern length is correct and does not
 *              contain any white space
 */
int Read_Pattern (Circuit *cut)
{
  g_pattern_sim = 1;

  if (!g_pattern_left) return 0;

  for (int i=0; i<cut->num_input; i++)
  {
    int ch = getc (stdin);

    if (ch == EOF || ch == '\n')
    {
      printf ("Error reading input patterns\n");
      exit (0);
    }

    if (ch == '1') ch = 1;
    else           ch = 0;

    int input_gate = cut->input_list[i];
    node_value[input_gate] = (node_value[input_gate] << 1)+ch;
  }
  getc (stdin);  // fetch the EOL character

  if (first_time)
  {
    first_time = 0;
    for (int i=0; i<cut->num_input; i++)
    {
      int input_gate = cut->input_list[i];

      node_value[input_gate] <<= 1;
      if (node_value[input_gate] & 2)
        node_value[input_gate] ^= 1;
    }
  }

  return g_pattern_left--;
}



//
// the following gate evalution routines are used for running on the
// workstation.  They are designed differently from the conventional
// and pattern parallel version so that it could be shared between
// FaultFreeSimulate() and Checkobs().
//

inline int ws_g_buff(Circuit *cut, int gate)
{
  return node_value[cut->gate_list[gate].in_node[0]];
}

inline int ws_g_not (Circuit *cut, int gate)
{
  return ~node_value[cut->gate_list[gate].in_node[0]];
}

inline int ws_g_and (Circuit *cut, int gate)
{
  int value = node_value[cut->gate_list[gate].in_node[0]];
  for (int j=1; j<cut->gate_list[gate].n_fan_in; j++)
    value &= node_value[cut->gate_list[gate].in_node[j]];
  return value;
}

inline int ws_g_nand(Circuit *cut, int gate)
{
  int value = node_value[cut->gate_list[gate].in_node[0]];
  for (int j=1; j<cut->gate_list[gate].n_fan_in; j++)
    value &= node_value[cut->gate_list[gate].in_node[j]];
  return ~value;
}

inline int ws_g_or  (Circuit *cut, int gate)
{
  int value = node_value[cut->gate_list[gate].in_node[0]];
  for (int j=1; j<cut->gate_list[gate].n_fan_in; j++)
    value |= node_value[cut->gate_list[gate].in_node[j]];
  return value;
}

inline int ws_g_nor (Circuit *cut, int gate)
{
  int value = node_value[cut->gate_list[gate].in_node[0]];
  for (int j=1; j<cut->gate_list[gate].n_fan_in; j++)
    value |= node_value[cut->gate_list[gate].in_node[j]];
  return ~value;
}

inline int ws_g_xor (Circuit *cut, int gate)
{
  int value = node_value[cut->gate_list[gate].in_node[0]];
  for (int j=1; j<cut->gate_list[gate].n_fan_in; j++)
    value ^= node_value[cut->gate_list[gate].in_node[j]];
  return value;
}

inline int ws_g_xnor(Circuit *cut, int gate)
{
  int value = node_value[cut->gate_list[gate].in_node[0]];
  for (int j=1; j<cut->gate_list[gate].n_fan_in; j++)
```

```
    value ^= node_value[cut->gate_list[gate].in_node[j]];
  return ^value;
}


inline FaultFreeSim (Circuit *cut)
{
  for (int i=0; i<cut->num_gate; i++)
  {
    switch (cut->gate_list[i].gate_type)
    {
      case g_INPT: break;

      case g_BUFF: node_value[i] = ws_g_buff(cut, i); break;
      case g_NOT : node_value[i] = ws_g_not (cut, i); break;
      case g_AND : node_value[i] = ws_g_and (cut, i); break;
      case g_NAND: node_value[i] = ws_g_nand(cut, i); break;
      case g_OR  : node_value[i] = ws_g_or  (cut, i); break;
      case g_NOR : node_value[i] = ws_g_nor (cut, i); break;
      case g_XOR : node_value[i] = ws_g_xor (cut, i); break;
      case g_XNOR: node_value[i] = ws_g_xnor(cut, i); break;
    }

    //printf ("gate %4d : %d\n", i, node_value[i] & 3);
  }
}


/*
 * Mark_detected
 *
 * marks a tree of faults as detected by implication recursively
 *
 */
void Mark_detected (Fault *fault)
{
  fault->flag |= fd_IMPLIED;

  if (fault->implied1)
    if (fault->implied1->flag == fd_UNDETECTED)
    Mark_detected (fault->implied1);

  if (fault->implied2)
    if (fault->implied2->flag == fd_UNDETECTED)
    Mark_detected (fault->implied2);

  return;
}


// It is assumed that the fault families contains at most one level of
// gate.  A multiple level fault family requires more complicated
// programming, which implies more time is need for
// checking observability.

inline boolean Checkobs(Circuit *cut, int gate)
{
  if (!gate) return true;

  int fault_value;

  switch (cut->gate_list[gate].gate_type)
  {
    case g_INPT: break;

    case g_BUFF: fault_value = ws_g_buff(cut, gate); break;
    case g_NOT : fault_value = ws_g_not (cut, gate); break;
    case g_AND : fault_value = ws_g_and (cut, gate); break;
    case g_NAND: fault_value = ws_g_nand(cut, gate); break;
    case g_OR  : fault_value = ws_g_or  (cut, gate); break;
    case g_NOR : fault_value = ws_g_nor (cut, gate); break;
    case g_XOR : fault_value = ws_g_xor (cut, gate); break;
    case g_XNOR: fault_value = ws_g_xnor(cut, gate); break;
  }

  return ((fault_value & 1) != (node_value[gate] & 1));
}


int Check_Fault (Circuit *cut, Fault *fault)
{
  if (fault->flag != fd_UNDETECTED) return 0;

  int from_gate = cut->wire_list[fault->wire_num].from_gate;
  int to_gate   = cut->wire_list[fault->wire_num].to_gate;


  boolean answer;

  switch (fault->type)
  {
    case f_SA0:
    {
      if (node_value[from_gate] & 1)
      {
        if (cut->gate_list[from_gate].n_fan_out > 1)
        {
          node_value[from_gate] &= ~1;
          answer = Checkobs(cut, to_gate);
          node_value[from_gate] |= 1;
        }
        else answer = true;
      }
      else answer = false;
    }
    break;

    case f_SA1:
    {
      if (!(node_value[from_gate] & 1))
      {
        if (cut->gate_list[from_gate].n_fan_out > 1)
        {
          node_value[from_gate] |= 1;
          answer = Checkobs(cut, to_gate);
          node_value[from_gate] &= ~1;
        }
        else answer = true;
      }
      else answer = false;
    }
    break;

    case f_SR:
    {
      if (node_value[from_gate] & 1 &&
        !(node_value[from_gate] & 2))
      {
        if (cut->gate_list[from_gate].n_fan_out > 1)
        {
          node_value[from_gate] &= ~1;
          answer = Checkobs(cut, to_gate);
          node_value[from_gate] |= 1;
        }
        else answer = true;
      }
      else answer = false;
    }
    break;

    case f_SF:
    {
      if (node_value[from_gate] & 2 &&
        !(node_value[from_gate] & 1))
      {
        if (cut->gate_list[from_gate].n_fan_out > 1)
        {
          node_value[from_gate] |= 1;
          answer = Checkobs(cut, to_gate);
          node_value[from_gate] &= ~1;
        }
        else answer = true;
      }
      else answer = false;
    }
    break;

    case f_SO:
    {
      if ( (node_value[from_gate] & 1) &&
         ( ( (node_value[to_gate] >> 1) ^
           (node_value[to_gate]) ) & 1) )
      {
        node_value[from_gate] &= ~1;
        answer = Checkobs(cut, to_gate);
        node_value[from_gate] |= 1;
      }
      else answer = false;
    }
    break;

    case f_PO:
    {
      if (!(node_value[from_gate] & 1) &&
         ( ( (node_value[to_gate] >> 1) ^
           (node_value[to_gate]) ) & 1) )
      {
        node_value[from_gate] |= 1;
        answer = Checkobs(cut, to_gate);
        node_value[from_gate] &= ~1;
      }
      else answer = false;
    }
    break;

    case f_iSO:
    {
      if (node_value[from_gate] & 2 &&
        !(node_value[from_gate] & 1))
      {
        answer = true;
      }
      else answer = false;
    }
    break;

    case f_iPO:
    {
      if (node_value[from_gate] & 1 &&
        !(node_value[from_gate] & 2))
      {
        answer = true;
      }
```

```
    else answer = false;
  }
  break;
}

//printf ("The fault is %sdetected\n",answer?"":"un");
return answer;
}


inline RemoveFaultFromFamily (Circuit *cut, FaultFamily *car)
{
  Fault *mcar = NULL;
  Fault *fcar = car->Member;
  car->Member = NULL;

  for (; fcar != NULL; fcar = fcar->next)
  {
    if (Check_Fault (cut, fcar))
    {
      Mark_detected (fcar);
      fcar->flag = fd_DETECTED;
    }

    if (fcar->flag == fd_UNDETECTED)
    {
      if (!car->Member) car->Member = fcar;
      else        mcar->next = fcar;
      mcar = fcar;
    }
  }
  if (mcar) mcar->next = NULL;
}


// Prepare_heap

inline int Prepare_heap (Circuit *cut,
            FaultFamily *ff_head, FaultFamily *sim_ff_head)
{
  int ff_insim=0;
  int ff_total=0;

  FaultFamily *acar = sim_ff_head;

  for (FaultFamily *car = ff_head->next;
     car != NULL; car = car->next, ff_total++)
  {
    if (car->type != (node_value[car->gate_num] & 1))
    {
      // first check special primary-output condition

      if (cut->gate_list[car->gate_num].n_fan_out == 0)
      {
        RemoveFaultFromFamily(cut, car);

        if (!car->Member)
        {
          if (car->prev) car->prev->next = car->next;
          if (car->next) car->next->prev = car->prev;
        }
      }
      else
      {
        boolean flag = false;
        for (Fault *fcar = car->Member;
           fcar != NULL; fcar = fcar->next)
        {
          if (flag = Check_Fault (cut, fcar)) break;
        }

        if (flag)
        {
          // add this family to simulation list
          acar->nextinsim = car;
          acar = car;

          ff_insim++;
        }
      }
    }
  }

  acar->nextinsim = NULL;

  return ff_insim;
}


////////////////////////////////////////////////////////////////
// this routine sorts the first perfParams.processors sim_ff_head
// elements in gate evaluation order.
// note that the routine assumes that pointer and intergers are both
// 32bit.  Also, it is assumed that the fault families were allocated
// as an array
////////////////////////////////////////////////////////////////
void Sort_FF_List (FaultFamily *sim_ff_head)
{
  new_heap();

  FaultFamily *car = sim_ff_head->nextinsim;

  for (int i=0; i<perfParams.processors; i++)
  {
    add_heap((int) car);
    car = car->nextinsim;
    if (!car) break ;
  }

  FaultFamily *head, *pop_car;
  FaultFamily *end = car;
  int ch;

  head = (FaultFamily*) pop_heap();
  car = head;
  while ((ch = pop_heap()) != -1)
  {
    pop_car = (FaultFamily *) ch;
    car->nextinsim = pop_car;
    car = pop_car;
  }
  car->nextinsim = end;

  sim_ff_head->nextinsim = head;
}


void Add_ff_to_heap(FaultFamily *sim_ff_head)
{
  new_heap();

  FaultFamily *car = sim_ff_head->nextinsim;

  for (int i=0; i<perfParams.processors; i++)
  {
    //printf ("  ff(%d/%d) added to heap\n",car->gate_num,car->type);
    event_list[car->gate_num] = 3;  // 2 = ff
    add_heap(car->gate_num);

    car = car->nextinsim;
    if (!car) return;
  }
}

inline boolean RoleCall (int *PE_num, cboolean *collect)
{
  static cint *temp = new cint(1, ALIGNED);
  boolean flag;
  int PE_number = *PE_num;

  if (PE_number >= 0)
  {
    // speed up trick: check continuous detection
    flag = true;
    temp->operate (0, op_m, shiftright);  //write y

    collect->operate (0, op_ybar | op_mbar, busaccess, flag);
    if (!flag)
    {
      *PE_num = PE_number+1;
      collect->operate (0, op_yzero, grouperite);
      temp->operate (0, op_y, grouperite);
      return true;
    }
  }
  else
  {
    // first time calling RoleCall
  }

  // check if any more PE contains a 1
  flag = true;
  collect->operate (0, op_m, writey);
  collect->operate (0, op_ybar, busaccess, flag);
  if (flag)
  {
    return false;
  }

  // begin binary search
  PE_number = 0;
  for (int i = (PEid.bits-1); i >= 0; i--)
  {
    flag = true;
    // check lower half of PEs
    PEid.operate (i, op_ybar | op_m, busaccess, flag);

    PE_number <<= 1;
    if (!flag)
    {
      // have detection in lower half of selected PEs
      PEid.operate (i, op_y & op_mbar, writey);
    }
    else
    {
      // have no detection in lower half of selected PEs
      PE_number++;
    }
  }

  // go on to next PE with fault
  collect->operate (0, op_yzero, grouperite);
  temp->operate (0, op_y, grouperite);
  *PE_num = PE_number;
```

```
  return true;
}

void Drop_Fault (Circuit *cut, FaultFamily *ff_head,
        int maxff, FaultFamily *sim_ff_head)
{
  cboolean collect;

  // collect all the primary output nodes into one cboolean

  collect.operate (0, op_zero, writex);

  for (int i=0; i<cut->num_output; i++)
  {
    int out_node = cut->wire_list[cut->output_list[i]].from_gate;

    cboolean *current = Obtain_Node_f_mapper (out_node);

    if (current)
    {
      if (node_value[out_node] & 1)
      {
        current->operate (0, op_mbar | op_x, writex);
      }
      else
      {
        current->operate (0, op_m | op_x, writex);
      }
    }

  }
  collect.operate (0, op_x, grouperite);

  // note that in a pattern parallel case,
  // the PEs that simulated the same faults
  // should also collect their result into one PE.

  // after collecting, extract the list of PEs that are detected.
  // search for the fault family that this PE represents.
  // check individual members of this family for triggering condition

  FaultFamily *car = sim_ff_head->nextinsim;
  int counter = 0;
  int count_ff_triggered = 0;

  int PE_number = -1;
  boolean flag;
  //cint temp(1,ALIGNED);
  for (;;)
  {
    flag = RoleCall (&PE_number, &collect);

    if (flag)
    {
      // this check shouldn't be necessary
      if (PE_number >= maxff)
      {
        printf ("PE_number(%d) > maxff(%d)!\n", PE_number, maxff);
        goto DONE_DROPPING;
      }

      STARTCLOCK;
      for (; counter < PE_number; counter++) car = car->nextinsim;

      RemoveFaultFromFamily(cut, car);

      if (!car->Member)
      {
        if (car->prev) car->prev->next = car->next;
        if (car->next) car->next->prev = car->prev;
      }
      STOPCLOCK;

      count_ff_triggered++;
      continue;
    }
    else
    {
      DONE_DROPPING:
      for (; counter < maxff; counter++)
        car = car->nextinsim;

      sim_ff_head->nextinsim = car;
      break;
    }
  }
}



/*
 * Simulate
 *
 */
inline int Simulate (Circuit *cut, FaultFamily *ff_head)
{
  STARTCLOCK;
  FaultFreeSim (cut);

  // sim_ff_head is used to store the list of fault family
  // need to be simulated.  Keep in mind that it is a dummy
  FaultFamily sim_ff_head;

  int maxff = Prepare_heap (cut, ff_head, &sim_ff_head);
  STOPCLOCK;

  #ifdef PEUTIL
  useful_PE += maxff;
  #endif

  while (maxff)
  {
    #ifdef PEUTIL
    used_PE++;
    #endif

    STARTCLOCK;
    Free_All_Node ();

#ifdef FFSDFS
    Sort_FF_List(&sim_ff_head);
#endif
    Add_ff_to_heap(&sim_ff_head);
    STOPCLOCK;

    FaultFamily *car = sim_ff_head.nextinsim;

    // setup y-register
    cboolean token;
    fault_free.operate (0, op_m, writey);
    token.operate (0, op_y, grouperite);

    int j;
    while ((j = pop_heap()) >= 0)
    {
      //printf ("gate %d poped from heap\n",j);
      n_gate_eval++;

      if (event_list[j] & 1)
      {
        //printf ("   simulating gate %d\n", j);
        Gate_Eval (cut, j);
      }

      boolean propagate = true;
      if (event_list[j] & 2)
      {
        cboolean *current = Obtain_Node_w_Loading (j);

        token.operate (0, op_m, writey);
        current->operate (0, op_y ^ op_m, grouperite);
        current->operate (0, op_y, shiftright);
        token.operate (0, op_y, grouperite);
        car = car->nextinsim;
      }

      if (event_list[j] == 1)
      {
        // no fault insertion, just simulation.
        // so determine fault propagation

        cboolean *current = Obtain_Node (j);
        boolean flag=true;

        if (node_value[j] & 1)
          current->operate (0, op_m, busaccess, flag);
        else
          current->operate (0, op_mbar, busaccess, flag);

        propagate = !flag;
      }

      event_list[j] = 0;


      // update heap
      if (propagate)
      {
        for (int k=0; k<cut->gate_list[j].n_fan_out; k++)
        {
          int out_node = cut->wire_list[
                  cut->gate_list[j].fan_out[k]
                ].to_gate;

          if (!event_list[out_node])
          {
            add_heap(out_node);
          }
          event_list[out_node] |= 1;
        }
      }
    }

    // printf ("ending simulation ...\n");
    // fflush(stdout);

    if (maxff >= perfParams.processors)
    {
      Drop_Fault (cut, ff_head, perfParams.processors, &sim_ff_head);
      maxff -= perfParams.processors;
    }
    else
    {
      Drop_Fault (cut, ff_head, maxff, &sim_ff_head);
      maxff = 0;
```

```
    }
  }
}


/*
 * Fault_Sim
 *
 * The main interface of this module.
 * Calling this routine would cause the program to start reading
 * in test patterns and arrange the faults for simulation.
 *
 * This routine implements the sequential PPSFP algorithm.
 *
 */

#define PRINT_PROCESS {                 \
  int  n_ss, n_d, n_so;                 \
  Count_Fault_List(fault_list,&n_ss, &n_d, &n_so);   \
  printf ("%8d %7.2f %7.2f %7.2f\n",    \
       g_total_pattern - g_pattern_left,    \
       (g_total_ss = n_ss)*100.0/g_total_ss,   \
       (g_total_d  = n_d )*100.0/g_total_d ,   \
       (g_total_so = n_so)*100.0/g_total_so    \
       );                               \
}


int  Fault_Sim(Circuit *cut, Fault *fault_list, int  n_pattern)
{
  Init_Sim (cut, n_pattern);

  // ff_head is used as dummy head
  FaultFamily ff_head;
  ff_head.next = Gen_Fault_Family (cut, fault_list);
  ff_head.next->prev = &ff_head;

  #ifndef PEUTIL
  printf ("%9s%10s%10s%10s\n","Pattern","%i","GD","SO");
  PRINT_PROCESS;
  #else
  printf ("%9s%16s%8s\n","Pattern","time","util ");
  printf ("%8d %15.9f %7.2f%\n",0.0,0.0);
  #endif

  int counter = 0;

  STOPCLOCK;

  while (Read_Pattern (cut))
  {
    // printf ("pattern %d\n", ++counter);
    Simulate (cut, &ff_head);

    STARTCLOCK;
    #ifndef PEUTIL
    PRINT_PROCESS;
    #else
    CALCCLOCK;
    printf ("%8d %15.9f %f\n",
       g_total_pattern - g_pattern_left,
       cvar.time/1000000000.0 + STOPWATCH_TIME/1000.0,
       useful_PE * 100.0 / (used_PE * perfParams.processors) );
    #endif
    STOPCLOCK;

    if (!ff_head.next) break;
  }

  #ifdef PEUTIL
  printf ("PE utilization = %6.2f (%d/%d)\n",
    useful_PE * 100.0 / (used_PE * perfParams.processors) ,
    useful_PE, used_PE * perfParams.processors);
  #endif

  printf ("number of gate eval = %d (%d PE)\n",
     n_gate_eval * perfParams.processors, perfParams.processors);
  PRINTCLOCK;
}

// end of simulate_fps.C
```

A.5 Hybrid Fault Simulator

```
/*
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Parallel Fault Simulators on the C*RAN Architecture
 *
 * Copyright (c) 1997, 1998 by Albert L.-C. Hsung, Edmonton, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * simulate_hps.C
 *
 * Purpose :  This file contains the routines for performing hybrid
 *            fault simulation.
 *
 * Compile Options :
 *   CR_PURE  - use the cran_mapper module
 *   FPSDFS   - enable depth first search fault grouping
 *   DYNAMICH - enable Dynamic Hybrid fault simulation
 *   PEUTIL   - measure PE utilization ratio
 */

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <ctype.h>
#include <math.h>
#include <cran.h>
#include <stopwatch.h>

#include "structure.h"
#include "prototype.h"
#include "heap.h"

#ifdef CR_PURE
#include "cran_mapper.h"
#endif


extern int NumPartition;    // number of C*RAN partitions

int  g_total_pattern;
int  g_pattern_left;
int  g_num_int ;
int  g_pattern_sim;       // number of pattern in simulation
int  g_n_input;

// C*RAN partitioning variables
int  g_n_part;
int  PE_per_part;

#ifdef FPSDFS
int  *part_map;           // maps partitions to gates
#endif
cint *part_mask;          // part_mask : an array of masks,
                          //   where each mask selects one partition
                          // PE_mask : select 1 PE in each partition
cboolean PE_mask_init;    // initial position of PE_mask
cboolean or_mask;
cboolean result;


cint *pattern;
cint *good_value;
cint *wire_value;
char *last_pat;   // store the last pattern


#ifdef DYNAMICH
int Dn_gate_input;
int Dn_triggered;
int Dn_gate_eval;
int Dn_MaxPart;
int Dn_MinPart;
#endif

#ifdef PEUTIL
int useful_PE;
int used_PE;
cboolean count_PE;

double util_time;
double stop_time;
#define STARTUTIL stop_time = count_PE.time;
#define ENDUTIL   util_time += count_PE.time - stop_time;
#endif


/* the following variables are used for transition faults */
char  first_time;       /* indicates first set of patterns */

/* the following global variables are made global to enhance speed */
static Circuit *g_cut;
char  *event_list;      /* if a gate is affected by a fault, the
                           event_list of that gate is set */


/* function prototypes */
void SetPartition(int n_part);
void Mark_detected (Fault *fault);
int  Good_Sim (Circuit *cut);
void Bad_Sim (Circuit *cut);
void cintcpy (cint &a, cint &b, int offset);
```

```
inline void Gate_Map (Circuit *cut, int j);


/////
// the caller of SetPartition is responsible for providing a correct
// n_part a correct n_part should be a power of 2
/////
void SetPartition(int n_part)
{
  if (n_part == g_n_part) return;
  g_n_part = n_part;


  if (part_mask) delete part_mask;
  part_mask = new cint(g_n_part,ALIGNED);

  if (!part_mask)
  {
    fprintf (stderr, "cannot allocate enough memory for part_mask\n");
    exit(1);
  }

  if (g_n_part == 1)
  {
    part_mask->operate (0, op_one, groupwrite);
    PE_mask_init.operate (0, op_one, shiftright);
    PE_mask_init.operate (0, op_ybar, groupwrite);
  }
  else
  {
    // setup part_mask;
    for (int i=0; i<g_n_part;i++)
    {
      part_mask->operate (i, op_one, writey);
      for (int j=(g_n_part>>1), k=PEid.bits-1; j>0; j>>=1, k--)
      {
        if (i&j) PEid.operate (k, op_a, writea);
        else     PEid.operate (k, op_abar, writea);

        PEid.operate (0, op_a & op_y, writey);
      }
      part_mask->operate(i,op_y, groupwrite);

      //printf ("part_mask[%d] ",i);
      //part_mask->PrintPEres(i);
    }

    // setup PE_mask_init
    //
    // PEid bits - nMSB = 00110011 (assuming 8 PEs 4 partition)
    //

    int nMSB = 0;
    for (int j=1; j != g_n_part; j<<=1) nMSB++;
    PEid.operate (PEid.bits-nMSB, op_abar, writea);
    PE_mask_init.operate (0, op_a, shiftright);
    PE_mask_init.operate (0, op_a & op_y, groupwrite);
  }

  //printf ("PE_mask_init ");
  //PE_mask_init.PrintPEres(0);
  //printf ("\n");

  PE_per_part = NUM_PE / n_part;
}


// assuming a.bits < b.bits, cpy from b to a
inline void cintcpy (cint &a, cint &b, int offset)
{
  int i;
  for (i=offset; i< a.bits; i++)
  {
    b.operate (i, op_a, writea);
    a.operate (i, op_a, groupwrite);
  }
}


/*
 * Init_Sim
 *
 * Initialize the simulation
 *
 */
void Init_Sim (Circuit *cut, int  n_pattern)
{
    /*
    struct rlimit rlp;
    getrlimit (RLIMIT_DATA, &rlp);
    printf ("soft limit = %d\n", (int) rlp.rlim_cur);
    printf ("hard limit = %d\n", (int) rlp.rlim_max);
    getrlimit (RLIMIT_STACK, &rlp);
    printf ("soft limit = %d\n", (int) rlp.rlim_cur);
    printf ("hard limit = %d\n", (int) rlp.rlim_max);
    */

  first_time = 1;

  g_cut       = cut;
  g_pattern_left  = n_pattern;
  g_total_pattern = n_pattern;
  g_n_input     = cut->num_input;

  event_list  = (char *) malloc (cut->num_gate);
  memset (event_list, 0, cut->num_gate);

  last_pat    = (char *) malloc (g_n_input);
  pattern     = new cint (g_n_input, ALIGNED);
  good_value  = new cint (cut->num_gate+1, ALIGNED);

  #ifdef CM_PURE
  wire_value = good_value;
  Init_CRAM_Mapper (cut, good_value);
  #else
  wire_value = new cint (cut->num_gate+1, ALIGNED);
  #endif

  g_n_part = 0;
  part_mask = NULL;

  #ifndef DYNAMICM
  SetPartition (NumPartition);
  #endif

  ini_heap();

  #ifdef FPSDPS
  part_map = (int *) malloc (sizeof(int) * g_cut->num_gate);
  if (!part_map)
  {
    fprintf (stderr, "cannot allocate for part_map\n");
    exit (1);
  }
  #endif

  #ifdef PEUTIL
  useful_PE = used_PE = 0;
  util_time = 0;
  #endif
}



/*
 * Read_Pattern
 *
 * Read PE_per_part patterns from input and store them into the
 * pattern data structure
 *
 * assumption : assume the input pattern length is correct and
 *              does not contain any white space
 */
int  Read_Pattern (Circuit *cut)
{
  int  ch;
  int  count_char=0;
  int  count_pattern=0;

  if (!g_pattern_left) return 0;

  count_pattern = count_char = 0;

  if (!first_time)
  {
    // copy last pattern
    for (int i=0; i<g_n_input; i++)
    {
      if (last_pat[i] == '1')
        pattern->operate (i, op_one, groupwrite);
      else
        pattern->operate (i, op_zero, groupwrite);
    }
    // set mask to second PE
    PE_mask_init.operate (0, op_a, shiftright);
  }
  else
  {
    // first time -> setup a full 1 mask
    PE_mask_init.operate(0, op_one, writey);
  }

  // fill up the pattern storage area
  while ((ch = getc(stdin)) != EOF)
  {
    if (ch == '\n')
    {
      //printf ("count_pattern %d\n",count_pattern);

      count_pattern++;
      g_pattern_left--;

      //else continue;
      if (first_time)
      {
        first_time = 0;
        // setup PE_mask at second PE
        PE_mask_init.operate (0, op_a, shiftright);
      }
      pattern->operate (0, op_y, shiftright);

      if (!g_pattern_left) break;

      // we only have so many PEs
```

```
    if (count_pattern == (PE_per_part-1)) break;

  }
  else
  {
    if (isspace (ch)) continue;
    if (count_char >= (PE_per_part * g_n_input)) break;

    last_pat[count_char%g_n_input] = (char) ch;

    if (ch == '1') pattern->operate (0, op_one, writex);
    else       pattern->operate (0, op_zero, writex);
    pattern->operate (count_char++%g_n_input, 0xe2, groupwrite);

  }
}

if (count_pattern * g_n_input != count_char)
{
  printf ("Error reading input patterns\n");
  exit (0);
}


// set up the masks
// remember that # patterns * #PEs per partition = 1,
// because the first PE is in special use
if (count_pattern == PE_per_part-1)
{
  // all PEs are active
  PE_mask_init.operate(0, op_n, writex); // do not want first PE
  or_mask.operate(0, op_xbar, groupwrite);
}
else
{
  // only PEs with pattern are active
  // op_y points to the first unused PE(s)
  or_mask.operate (0, op_y, writex);
  for (int i = count_pattern; i<(PE_per_part-1); i++)
  {
    or_mask.operate (0, op_y | op_x, writex);
    or_mask.operate (0, op_y, shiftright);  // write y
  }
  PE_mask_init.operate(0,op_n|op_x, writex); // do not want first PE
  or_mask.operate (0, op_xbar, groupwrite);  // y xbar
}

g_pattern_sim = count_pattern;
return count_pattern;
}


/*
 * Mark_detected
 *
 * marks a tree of faults as detected by implication recursively
 *
 */
void Mark_detected (Fault *fault)
{
  fault->flag |= fd_IMPLIED;

  if (fault->implied1)
    if (fault->implied1->flag == fd_UNDETECTED)
    Mark_detected (fault->implied1);

  if (fault->implied2)
    if (fault->implied2->flag == fd_UNDETECTED)
    Mark_detected (fault->implied2);

  return;
}


/*
 * CreateMask
 *
 * CreateMask is used for creating a bit mask in cram,
 * where a 1 indicates a triggering and 0 otherwise.  If
 * a non-zero cboolean is obtained, meaning that a fault
 * is triggered, then true will be returned, otherwise,
 * false is returned.  The resulting mask is stored in
 * the parameter cboolean mask
 *
 */
bool CreateMask(Circuit *cut, Fault *fault, cint *mask)
{
  if (fault->flag != fd_UNDETECTED) return false;

  int  fault_node  = cut->wire_list[fault->wire_num].node_num;
  int  from_gate   = cut->wire_list[fault->wire_num].from_gate;
  int  to_gate     = cut->wire_list[fault->wire_num].to_gate;
  int  gate_output = cut->gate_list[to_gate].out_node;

  bool checkPE = to_gate && (cut->gate_list[to_gate].n_fan_in > 1);


  switch (fault->type)
  {
    /*********************
     * if value of wire = 1, fault is triggered
     *********************/
    case f_SA0:
    {
      good_value->operate (fault_node, op_n, writex);
      mask->operate (0, op_x, groupwrite);
    }
    break;


    /*********************
     * if value of wire = 0, fault is triggered
     *********************/
    case f_SA1:
    {
      good_value->operate (fault_node, op_nbar, writex);
      mask->operate (0, op_x, groupwrite);
    }
    break;


    /*********************
     * triggered if there is a 0->1 transition
     * and there is a transition at output
     * an intermediate PO of OR gate is the same as SR at output
     *********************/
    case f_SR :
    case f_iPO:
    {
      good_value->operate (fault_node, op_n, shiftright);
      good_value->operate (fault_node, op_ybar&op_n, writex);
      mask->operate (0,op_x, groupwrite);
    }
    break;


    /*********************
     * triggered if there is a 1->0 transition
     * and there is a transition at output
     * an intermediate NO of AND gate is the same as SF at output
     *********************/
    case f_SF :
    case f_iNO:
    {
      good_value->operate (fault_node, op_n, shiftright);
      good_value->operate (fault_node, op_y&op_nbar, writex);
      mask->operate(0,op_x,groupwrite);
    }
    break;


    /*********************
     * a NO fault is triggered if there is a transition at
     * the gate output and the faulty input is 1
     *********************/
    case f_NO :
    {
      good_value->operate (gate_output, op_n, shiftright);
      good_value->operate (gate_output, op_y^op_n, writex);
      good_value->operate (fault_node, op_x&op_n, writex);
      mask->operate(0,op_x,groupwrite);
    }
    break;


    /*********************
     * a PO fault is triggered if there is a transition at
     * the gate output and the faulty input is 0
     *********************/
    case f_PO :
    {
      good_value->operate (gate_output, op_n, shiftright);
      good_value->operate (gate_output, op_y^op_n, writex);
      good_value->operate (fault_node, op_x&op_nbar, writex);
      mask->operate(0,op_x,groupwrite);
    }
    break;

    default: return false;
  }


  if (checkPE)
  {
    #ifdef CN_PURE
    cboolean *temp = Obtain_Node(fault_node);
    good_value->operate (fault_node, op_n, writex);
    temp->operate (0, op_xbar, groupwrite);

    Gate_Eval (cut, to_gate);

    temp = Obtain_Node(gate_output);
    good_value->operate (gate_output, op_n, writex);
    temp->operate (0, op_n^op_x, writey);
    Free_All_Node ();

    #else
    // insert fault, create PEffect mask, then clean up
    wire_value->operate (fault_node, op_nbar, groupwrite);
    Gate_Map (cut, to_gate);
    good_value->operate (gate_output, op_n, writex);
    wire_value->operate (gate_output, op_n^op_x, writey);

    // clean up wire_value
```

```
    wire_value->operate (gate_output, op_x, groupwrite);
    wire_value->operate (fault_node, op_nbar, groupwrite);
    #endif

    mask->operate (0, op_ykop_n, groupwrite); // collect mask
  }

  boolean all_zero = true;
  mask->operate (0, op_nbar, busaccess, all_zero);
  if (all_zero) return false;
  return true;
}

/*
 * Triggered
 *
 * For each family in the fault group, linearly search through
 * the member list to look for faults that are triggered.  If a
 * fault is triggered, then the TrigFault pointer is point to the
 * fault, and the resulting cboolean is used as a mask and pointed
 * to by the TrigFlag pointer.
 *
 * return false if no fault is triggered
 */
bool Triggered(Circuit *cut, FaultGroup *fg)
{
  bool triggered = false;

  for (int i=0; i<2; i++)
  {
    if (!fg->FF[i]) continue;

    bool local_triggered = false;
    cint *mask = new cint(1,ALIGNED);

    for (Fault *car = fg->FF[i]->Member; car != NULL; car = car->next)
    {
      // create the mask
      local_triggered = CreateMask(cut, car, mask);

      /*
      printf ("%s[%d] - ",fault_table[car->type],car->wire_num);
      printf ("%striggered ",local_triggered""_"not ");
      mask->PrintPError(0);
      */

      // if triggered, store information
      if (local_triggered)
      {
        fg->TrigFault[i] = car;
        fg->TrigFlag[i]  = mask;
        triggered |= local_triggered;
        break;
      }
    }

    if (!local_triggered)
    {
      fg->TrigFault[i] = NULL;
      fg->TrigFlag[i]  = NULL;
      delete mask;
    }
  }

  return triggered;
}


/*
 * Detect_Fault
 *
 * Starting from each fault pointed to be the TrigFault pointer,
 * look for further faults that are detected, mark them either
 * recursively or directly
 *
 * return true if any member of the group contains a detected
 * fault.  (which may be marked in this call, or marked by
 * implication by another call
 */
bool Detect_Fault(Circuit *cut, FaultGroup *fg, int partition)
{
  boolean detected = false;
  cint temp(1,ALIGNED);

  result operate(0, op_n, writex);
  part_mask->operate(partition, op_n & op_x, writex);
  temp operate(0,op_x,groupwrite);

  boolean undo = true;
  temp operate(0,op_xbar,busaccess,undo);

  #ifdef PEUTIL
  STARTUTIL
  if (undo)
  {
    cboolean temp2;

    temp2 operate (0, op_zero, groupwrite);
    for (int i=0; i<2; i++)
    {
      if (fg->TrigFault[i])
      {
        cint *mask = fg->TrigFlag[i];
        mask->operate (0, op_n, writex);
        temp2.operate (0, op_n | op_x, groupwrite);

        for (Fault *car = fg->TrigFault[i];
             car != NULL; car = car->next)
        {
          if (car->flag != fd_UNDETECTED)
          {
            detected = true;
            continue;
          }
          CreateMask(cut, car, mask);
          mask->operate (0, op_n, writex);
          temp2.operate (0, op_n | op_x, groupwrite);
        }
      }
    }

    temp2.operate (0, op_n, writex);
    part_mask->operate (partition, op_n & op_x, writex);
    or_mask.operate (0, op_n & op_x, writex);
    count_PE.operate (0, op_n | op_x, groupwrite);
  }
  ENDUTIL
  #endif

  if (undo)
  {

    for (int i=0; i<2; i++)
    {
      fg->TrigFault[i] = NULL;
      delete fg->TrigFlag[i];
      fg->TrigFlag[i] = NULL;
    }
    return false;
  }

  for (int i=0; i<2; i++)
  {
    //printf ("FG : %d-%d\n",fg->gate_num,i);

    if (!fg->TrigFault[i]) continue;

    bool checked = true;
    cint *mask = fg->TrigFlag[i];

    for (Fault *car = fg->TrigFault[i]; car != NULL; car = car->next)
    {
      if (car->flag != fd_UNDETECTED)
      {
        detected = true;
        continue;
      }

      bool triggered=true;
      if (!checked) triggered = CreateMask(cut, car, mask);
      else      checked = false;

      /*
      printf ("%s[%d] - ",fault_table[car->type],car->wire_num);
      printf ("%striggered ",triggered""_"not ");
      mask->PrintPError(0);
      */

      if (!triggered) continue;


      boolean undetected = true;
      temp operate(0, op_n, writex);
      mask->operate(0, "(op_x&op_n)", busaccess, undetected);


      /*
      printf ("   x - ");
      temp PrintPError(0);
      printf ("%sdetected\n",detected""_"not ");
      */

      if (!undetected)
      {
        Mark_detected(car);
        car->flag = fd_DETECTED;
        detected = true;
      }

      #ifdef PEUTIL
      STARTUTIL
      if (undetected)
      {
        mask->operate (0, op_n, writex);
        part_mask->operate(partition, op_n & op_x, writex);
        or_mask.operate (0, op_n & op_x, writex);
        count_PE.operate (0, op_n | op_x, groupwrite);
      }
      else
      {
        // search for the first PE that detects the fault
        // pre  : op_x = temp
        // post: temp2 will have the bits before this PE set to 1
        cboolean temp2;
```

```
      boolean mask;
      mask->operate (0, op_m & op_x, writey);

      temp2.operate (0, op_zero, groupwrite);
      smask.operate (0, op_xxx, groupwrite);

      for (int j=(PEid.bits-1); j>=0; j--)
      {
        boolean flag = true;
        PEid.operate (j, op_m, writex);
        PEid.operate (j, op_ybar | op_x, busaccess, flag);

        if (!flag)
        {
          // in lower half
          smask.operate (0, op_xbar & op_m, groupwrite);
          temp2.operate (0, op_y & op_xbar, writey);
        }
        else
        {
          smask.operate (0, op_xbar & op_m, writex);
          temp2.operate (0, op_m | op_x, groupwrite);
        }
      }
      temp2.operate (0, op_m, shiftright);  // include detection
      mask->operate (0, op_y & op_m, writex); // triggerings
      part_mask->operate(partition, op_m & op_x, writex);
      or_mask.operate (0, op_m & op_x, writex);
      count_PE.operate (0, op_m | op_x, groupwrite);

      /*
      printf ("\n");
      temp2.PrintPErow(0);
      mask->PrintPErow(0);
      count_PE.PrintPErow(0);
      */
    }
    ENDUTIL;
    #endif

  }

  fg->TrigFault[i] = NULL;
  fg->TrigFlag[i]  = NULL;
  delete mask;
 }

 return detected;
}


/*
 * CollectResult
 *
 * compare the primary output to the fault-free primary output
 * the result (boolean) will have PEs with detected fault set to 1
 */
void CollectResult(Circuit *cut)
{
 result.operate (0, op_zero, writex);
 for (int i=0; i<cut->num_output; i++)
 {
   int out_node = cut->wire_list[cut->output_list[i]].from_gate;

   #ifdef CK_FUNC
   boolean *temp = Obtain_Node_f_mapper (out_node);
   if (temp)
   {
     temp->operate (0, op_m, writey);
     good_value->operate(out_node, op_xi(op_m^op_y), writex);
   }
   #else
   wire_value->operate(out_node, op_m, writey);
   good_value->operate(out_node, op_xi(op_m^op_y), writex);
   #endif

 }
 or_mask.operate(0, op_m & op_x, writex);
 result.operate (0, op_x, groupwrite);
}


/*
 * FaultSession
 *
 * After good simulation, the list of fault-group is traversed
 * at each pass, a number of fault groups are inserted into
 * C*RAM for fault simulation.  After which, each of the
 * partition is checked to see if any Fault in the corresponding
 * fault-group is detected.  If so, the faults are dropped from
 * the fault-group list.
 *
 */

void FaultSession(Circuit *cut, FaultGroup *FG_head)
{
 int queue_count = 0;        // number of elements in queue
 FaultGroup *SimQueue[g_n_part];

 FaultGroup *fg_cur = FG_head->dfs_next;

 while (fg_cur != NULL)
 {
   new_heap();

   #ifdef FPGPS
   int cur_part = 0;
   #endif

   while (queue_count != g_n_part)
   {
     if (Triggered(cut, fg_cur))
     {
       SimQueue[queue_count++] = fg_cur;
       // add to heap
       event_list[fg_cur->gate_num] = 2;
       add_heap(fg_cur->gate_num);

       #ifdef FPGPS
       part_map[fg_cur->gate_num] = cur_part++;
       #endif
     }

     fg_cur = fg_cur->dfs_next;
     if (fg_cur == NULL) break;
   }

   if (queue_count == 0) break;

   #ifdef DYNASIM
   Dn_triggered += queue_count;
   #endif

   Bad_Sim(cut);

   #ifdef PEUTIL
   STARTUTIL;
   count_PE.operate (0, op_zero, groupwrite);
   ENDUTIL;
   #endif

   for (int i=0; i<queue_count; i++)
   {
     if (Detect_Fault(cut, SimQueue[i], i))
     {
       CleanFaultGroup(SimQueue[i]);
     }
   }

   #ifdef PEUTIL
   used_PE ++;
   useful_PE += count_PE.NumberOfOnes(0);
   #endif

   queue_count = 0;
 }
}


inline void g_inpt (int  j)
{
 /* load input pattern */
 pattern->operate    (g_cut->gate_list[j].input_num,op_m,writex);
 wire_value->operate(g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_buff (int  j)
{
 wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);
 wire_value->operate(g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_not  (int  j)
{
 wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);
 wire_value->operate(g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}

inline void g_and  (int  j)
{
 wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);

 int  i;
 for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
   wire_value->operate
     (g_cut->gate_list[j].in_node[i],op_xandm,writex);

 wire_value->operate(g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_nand (int  j)
{
 wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);

 int  i;
 for (i=1; i<g_cut->gate_list[j].n_fan_in; i++)
   wire_value->operate
     (g_cut->gate_list[j].in_node[i],op_xandm,writex);

 wire_value->operate(g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}


inline void g_or   (int  j)
```

```
{
  wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);

  int  l;
  for (l=1; l<g_cut->gate_list[j].n_fan_in; l++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[l],op_xorm,writex);

  wire_value->operate(g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_nor  (int  j)
{
  wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);

  int  l;
  for (l=1; l<g_cut->gate_list[j].n_fan_in; l++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[l],op_xorm,writex);

  wire_value->operate(g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}

inline void g_xor  (int  j)
{
  wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);

  int  l;
  for (l=1; l<g_cut->gate_list[j].n_fan_in; l++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[l],op_xorm,writex);

  wire_value->operate(g_cut->gate_list[j].out_node,op_x,groupwrite);
}

inline void g_xnor (int  j)
{
  wire_value->operate(g_cut->gate_list[j].in_node[0],op_m,writex);

  int  l;
  for (l=1; l<g_cut->gate_list[j].n_fan_in; l++)
    wire_value->operate
      (g_cut->gate_list[j].in_node[l],op_xorm,writex);

  wire_value->operate(g_cut->gate_list[j].out_node,op_xbar,groupwrite);
}

inline void Gate_Map (Circuit *cut, int j)
{
  switch (cut->gate_list[j].gate_type)
  {
    case g_INPT  : g_inpt(j); break;
    case g_BUFF  : g_buff(j); break;
    case g_NOT   : g_not (j); break;
    case g_AND   : g_and (j); break;
    case g_NAND  : g_nand(j); break;
    case g_OR    : g_or  (j); break;
    case g_NOR   : g_nor (j); break;
    case g_XOR   : g_xor (j); break;
    case g_XNOR  : g_xnor(j); break;
  }
}

/*
 * Bad_Sim
 *
 */
void Bad_Sim (Circuit *cut)
{
  int  j;    /* for loop counters */

  int  dirty_node[cut->num_gate];
  int  event_done = 0;

  int cur_part = 0;

  while ((j = pop_heap()) >= 0)
  {
    dirty_node[event_done++]=j;

    #ifdef DYNAMICH
    Dn_gate_eval++;
    #endif

    /////
    // gate evaluation
    /////
    if (event_list[j] & 1)
    {
      #ifdef CN_PURE
      Gate_Eval(cut, j); // gate evaluation routine in C*RAN Mapper
      #else
      Gate_Map (cut, j);
      #endif
    }

    boolean nopropagate = true;

    /////
    // fault insertion
    /////
    if (event_list[j] & 2)
    {
      #ifdef CN_PURE
      cboolean *temp = Obtain_Node_w_Loading (j);
      #endif

      #ifdef FPSDFS
      part_mask->operate(part_map[j], op_m, writey);
      #else
      part_mask->operate(cur_part, op_m, writey);
      cur_part++;
      #endif


      #ifdef CN_PURE
      temp->operate (0, op_m^op_y, groupwrite);
      #else
      wire_value->operate(j, op_m ^ op_y, groupwrite);
      #endif

      nopropagate = false;
    }

    /////
    // check for changes in output value
    /////
    if (event_list[j] == 1)
    {
      #ifdef CN_PURE
      cboolean *temp = Obtain_Node (j);
      temp->operate (0, op_m, writex);
      #else
      wire_value->operate (cut->gate_list[j].out_node,op_m,writex);
      #endif

      good_value->operate (cut->gate_list[j].out_node,~(op_x^op_m),
        busaccess,nopropagate);
    }

    event_list[j] = 0;

    if (!nopropagate)
    {
      /////
      // add event for further propagation
      /////

      for (int i=0;i<cut->gate_list[j].n_fan_out; i++)
      {
        int  nexte = cut->wire_list[
          cut->gate_list[j].fan_out[i] ].to_gate;

        if (!(event_list[nexte] & 1))
        {
          event_list[nexte] |= 1;
          add_heap(nexte);
        }
      }
    }
  }

  CollectResult(cut);

  // clean the wire_value for next time
  #ifdef CN_PURE
  Free_All_Node();
  #else
  for (j=0; j<event_done; j++)
  {
    good_value->operate (dirty_node[j], op_m, writex);
    wire_value->operate (dirty_node[j], op_x, groupwrite);
  }
  #endif
}


/*
 * Good_Sim
 */
int  Good_Sim (Circuit *cut)
{
  /* start going thru the circuit */
  for (int j=0; j<cut->num_gate; j++)
  {
    Gate_Map (cut, j);
  }

  #ifndef CN_PURE
  cintcpy(ogood_value, owire_value, 0);
  #endif

  return 0;
}

////////////////////////////////////////////////////
//
//  Functions for implementing Dynamic-Hybrid simulation
//
////////////////////////////////////////////////////

#ifdef DYNAMICH
```

```
double FindGST(Circuit *cut);
double FindCNT(Circuit *cut);
double FindBST(Circuit *cut);

double FindGST(Circuit *cut)
{
  Dn_gate_input = 0;

  for (int i=0; i<cut->num_gate; i++)
    Dn_gate_input += cut->gate_list[i].n_fan_in;

  return (double) Dn_gate_input * cut->num_gate;
}

double FindCNT(Circuit *cut)
{
  int needPE = 0;

  // number of wires that needs Primary Effect checking
  for (int i=0; i<cut->num_wire; i++)
  {
    if (cut->wire_list[i].to_gate)
    {
      int to_gate = cut->wire_list[i].to_gate;
      if (cut->gate_list[to_gate].n_fan_in > 1)
        needPE ++;
    }
  }

  double r_poffect = (double) needPE / cut->num_wire;
  double avg_poffect = ((double) Dn_gate_input/cut->num_gate) * 1 + 8;
  double avg_trigger = 3.0;

  return avg_trigger + (avg_poffect * r_poffect);
}

double FindBST(Circuit *cut)
{
  int passes = ((Dn_triggered-1)/g_n_part)+1;

  return (double) Dn_gate_eval / passes;
}

int AdjustPartition (double GST, double CNT, double BST,
        int D0, int D1, int T1)
{
  double cost_double;
  double cost_current;
  double cost_half;

  double D2 = (double)D1 * D1 / D0;
  double D15 = sqrt (D1 * D2);
  double D3 = D2 * D2 / D1;
  double D25 = sqrt (D2 * D3);

  double ratio = T1/D1;

  double T15 = D15 * ratio;
  double T2 = D2 * ratio;
  double T25 = D25 * ratio;

  // g_n_part = f

  cost_current = 2*GST + CNT*(D1 + D2) + BST*(T1 + T2)/g_n_part;

  if (g_n_part != Dn_MaxPart)
  {
    cost_double = GST + CNT*D1 + BST*T1/(g_n_part>>1);
  }
  else
  {
    cost_double = 2 * cost_current;
  }

  if (g_n_part != Dn_MinPart)
  {
    cost_half = 4*GST + CNT + (D1 + D15 + D2 + D25)
          + BST*(T1 + T15 + T2 + T25)/(g_n_part<<1);
  }
  else
  {
    cost_half = 2 * cost_current;
  }

  if (cost_half < cost_double)
  {
    if (cost_half < cost_current)
    {
      printf ("halving p : %d\n",
        perfParams::processors/(g_n_part << 1));
      return g_n_part << 1;
    }
  }
  else
  {
    if (cost_double < cost_current)
    {
      printf ("doubling p : %d\n",
        perfParams::processors/(g_n_part >> 1));
      return g_n_part >> 1;
    }
  }

  return g_n_part;
}

#endif

////////////////////////////////////////////////////

/*
 * Fault_Sim
 *
 * The main interface of this module.
 * Calling this routine would cause the program to start
 * reading in test patterns and arrange the faults for simulation.
 *
 * This routine implements the sequential PPSFP algorithm.
 *
 */
#define PRINT_PROCESS {                 \
  printf ("%8d %7.2f %7.2f %7.2f\n",    \
       g_total_pattern - g_pattern_left,    \
       (g_total_sa - n_sa)*100.0/g_total_sa,   \
       (g_total_d  - n_d )*100.0/g_total_d ,   \
       (g_total_so - n_so)*100.0/g_total_so    \
       );                               \
  fflush(stdout);                       \
}

int  Fault_Sim(Circuit *cut, Fault *fault_list, int  n_pattern)
{
  int  n_sa, n_d, n_so;

  Init_Sim (cut, n_pattern);

  // ff_head is used as dummy head
  FaultGroup *FG_head = InitFaultGroup(cut, fault_list);

  Count_Fault_List(fault_list,&n_sa, &n_d, &n_so);

  #ifndef PEUTIL
  printf ("%8s%8s%8s%8s\n","Pattern","SA","GD","SO");
  PRINT_PROCESS;
  #else
  printf ("%8s%16s%8s\n","Pattern","time","util.");
  printf ("%8d %15.9f %7.2f\n",0,0,0);
  #endif

  #ifdef DYNAMICH
  double GST = FindGST(cut);
  double CNT = FindCNT(cut);
  double BST;
  int D0;
  int D1 = n_sa + n_d + n_so;
  Dn_MaxPart = perfParams::processors/2;
  Dn_MinPart = 1;

  STOPCLOCK;
  SetPartition(Dn_MaxPart);
  #else
  STOPCLOCK;
  #endif

  while (Read_Pattern (cut))
  {
    Good_Sim (cut);

    FaultSession(cut, FG_head);

    #ifdef DYNAMICH
    D0 = D1;
    #endif

    STARTCLOCK;
    Count_Fault_List(fault_list,&n_sa, &n_d, &n_so);

    #ifndef PEUTIL
    PRINT_PROCESS;
    #else
    printf ("%8d %15.9f %f\n",
      g_total_pattern - g_pattern_left,
      (count_PE_time - util_time)/1000000000,
      useful_PE * 100.0 / (used_PE * perfParams::processors) );
    #endif

    if (FG_head->dfs_next == NULL) break;

    #ifdef DYNAMICH
    BST = FindBST(cut);
    D1 = n_sa + n_d + n_so;
    int n_part = AdjustPartition(GST, CNT, BST, D0, D1, Dn_triggered);
    STOPCLOCK;
    if (n_part != g_n_part) SetPartition(n_part);
    #else
    STOPCLOCK;
    #endif
  }
```

```
  #ifdef PEUTIL
  printf ("PE utilization = %6.2f (%d/%d)\n",
    useful_PE * 100.0 / (used_PE * perfParams::processors) ,
    useful_PE, used_PE * perfParams::processors);
  #endif
}


// end of simulate_hps.C
```